

VS

Procedure Language

Reference

5th Edition — January 1984
Copyright © Wang Laboratories, Inc., 1979
800-1205-05

WANG

WANG LABORATORIES, INC., ONE INDUSTRIAL AVENUE, LOWELL, MA 01851 • TEL. (617) 459-5000, TWX 710-343-6769 Telex 94-7421

PREFACE

This manual describes the current VS Procedure language interpreter and is designed for use with version 6.20 or later of the the VS Operating System. It is divided into two sections. The User Guide Section provides information for the programmer who is acquainted with programming concepts, but unfamiliar with VS Procedure language. Chapters 1 through 11, which comprise the User Guide, provide a sequential overview of the current Procedure Interpreter. The Reference Guide presents an alphabetical list of Procedure language statements.

Chapter 1 presents the elements of the language. This chapter describes when to use procedures, what procedures can do, and provides some general syntax information.

Chapter 2 discusses Procedure language terminology. Procedures are constructed by using a series of procedure statements, which consist of one or more verbs, expressions, and labels. This chapter defines each of these terms in detail.

Chapter 3 presents information about running programs with a procedure. This chapter introduces the RUN statement, and the RUN statement clauses, DISPLAY and ENTER.

Chapter 4 discusses methods of altering the flow of control within a procedure. Procedures are executed sequentially, unless a statement directs control to another portion of the procedure.

Chapter 5 contains information about System Call statements. System Call statements invoke system functions. These statements support tasks involved in both program and file processing. They can also be used to manipulate peripheral devices. Chapter 6 discusses the OPTIONS statement, as well as built-in functions.

Chapter 7 completes the discussion about the RUN statement. It fully discusses the RUN statement, and discusses in detail the USING, ERROR EXIT IS, and the CANCEL EXIT IS clauses. The USING statement is also discussed.

Chapter 8 presents the DECLARE and ASSIGN statements.

Chapter 9 discusses how to display and accept text, variables and constants on the workstation by using the MESSAGE and PROMPT statements.

Chapter 10 presents the Syntax Checker, which can be used as you would use a compiler in other languages. It will flag most of your syntax errors in a single run, without executing the procedure.

Chapter 11 discusses the Tracing facility, which creates a record of Procedure Interpreter activity.

Chapter 12 is the Reference chapter. This chapter presents each statement in alphabetical order. For each statement, the function, general format, syntax rules, general rules and an example is presented.

Appendix A presents Return Code values for commonly used VS Utilities and for the Procedure language verbs that return a code. Appendix B presents a list of incompatibilities between the new and old interpreter.

The following manuals contain helpful information related to VS Procedure language:

- VS File Management Utilities Reference 800-1308
- VS Program Development Tools 800-1307
- VS Programmer's Introduction 800-1101
- VS System Operations Guide 800-1102
- VS System Utilities Reference 800-1303

The following manuals discuss the use of Procedure language in telecommunications:

- 3271 Emulation User's Guide 800-1306
- VS General Purpose Asynchronous Communications Programmer's Guide 800-1325
- Teletypewriter Emulation User Guide 800-1314
- Batch Communications User Guide 800-1305

CONTENTS

PART I USER GUIDE

CHAPTER 1 ELEMENTS OF THE LANGUAGE

1.1	Procedures	1-1
1.2	Entering and Running Procedures	1-3
	Entering and Running a Procedure	
	from the PROC Program	1-3
	Entering and Running a Procedure	
	from the EDITOR	1-4
1.3	Procedure language Syntax	1-4

CHAPTER 2 PROCEDURE TERMINOLOGY

2.1	Verbs	2-1
2.2	Expressions	2-3
	Constants	2-3
	Variables	2-4
	Substrings	2-5
	Built-in Functions	2-5
	Backward References	2-6
	Operators	2-6
2.3	Labels	2-8

CHAPTER 3 RUNNING PROGRAMS WITH A PROCEDURE (RUN, DISPLAY AND ENTER)

3.1	Levels of Default	3-2
	Specifying the File, Library, and Volume	3-2
	Specifying the File and Library	3-2
	Specifying the File and Volume	3-3
	Specifying a File Name Only	3-3
3.2	The GETPARM Request	3-3
	GETPARMS	3-3
	Parameter Reference Name	3-4
	Keywords	3-4
3.3	Supplying Parameter Values With	
	DISPLAY and ENTER	3-4
	The ENTER Clause	3-5
	The DISPLAY Clause	3-7
3.4	Referencing With Labels	3-8
	Backward Reference	3-9
	Backward Reference to Individual Parameters	3-10
	Multiple Labels	3-11
3.5	Nested Procedures- Overriding with	
	DISPLAY and ENTER	3-11

CONTENTS (continued)

CHAPTER	4	FLOW OF CONTROL WITHIN A PROCEDURE	
	4.1	Terminating Procedure Execution	4-1
		The RETURN Statement	4-1
		The DESTROY Statement	4-3
	4.2	Controlling Conditional Execution	4-3
		The IF Statement	4-3
		The IF...Exists Statement	4-4
	4.3	Forward or Backward Branches: The GOTO Statement	4-5
CHAPTER	5	SYSTEM CALL STATEMENTS	
	5.1	Setting Default Values: The SET Statement	5-1
	5.2	The EXTRACT Statement	5-3
	5.3	Scratching a File or Library: The SCRATCH Statement .	5-5
	5.4	Renaming a File or Library: The RENAME Statement	5-6
	5.5	Protecting a File or Library: The PROTECT Statement .	5-6
	5.6	Queuing a Print File: The PRINT Statement	5-7
	5.7	Background Processing: The SUBMIT Statement	5-7
	5.8	Logically Mounting and Dismounting a Volume	5-10
		The MOUNT Statement	5-10
		The DISMOUNT Statement	5-10
	5.9	Logging Off: The LOGOFF Statement	5-10
CHAPTER	6	THE OPTIONS STATEMENT AND BUILT-IN FUNCTIONS	
	6.1	The OPTIONS Statement	6-1
	6.2	Built-in Functions	6-2
		&length	6-2
		&time	6-2
		&date	6-3
CHAPTER	7	RUNNING PROGRAMS AND PROCEDURES WITH A PROCEDURE	
	7.1	Unsuccessful Links- The ERROR EXIT IS Clause	7-1
	7.2	Cancelled Procedures- The CANCEL EXIT IS Clause	7-2
	7.3	Declaring the Formal and Actual Parameters	7-3
		Pass by Reference	7-3
		Pass by Value	7-3
		Pass by Value-Result	7-3
		The USING Clause and the USING Statement	7-4
CHAPTER	8	ASSIGNING VALUES & DEFINING VARIABLES: DECLARE & ASSIGN	
	8.1	Declaring Variables The DECLARE Statement	8-1
	8.2	Assigning Values: The ASSIGN Statement	8-1

CONTENTS (continued)

CHAPTER	9	DISPLAYING AND ACCEPTING TEXT, VARIABLES AND CONSTANTS ON THE WORKSTATION	
9.1		Displaying A Message on the Workstation:	
		The MESSAGE Statement	9-1
		Displaying Lines	9-1
		Creating Lines From Multiple Fields	9-2
		Video Attributes	9-2
		Using the ERASE Option to Build Messages	9-3
		Ringing the Workstation Bell	9-4
9.2		Soliciting A Response From the Workstation:	
		The PROMPT Statement	9-4
		Creating Modifiable Fields	9-5
		Receiving PF Key, Cursor Row and Cursor Column ...	9-6
		Tabs	9-6
CHAPTER	10	THE SYNTAX CHECKER	
10.1		Running the Syntax Checker	10-2
		Syntax Checking from the EDITOR	10-3
		Running the Syntax Checker From the PROC Program	10-5
CHAPTER	11	THE TRACING FACILITY	
11.1		How the Tracing Facility Functions	11-1
11.2		Running The Tracing Facility	11-1
		Tracing from a Procedure	11-2
		Running the Tracing Facility from the EDITOR	11-4
		Running the Tracing Facility as Part of the PROC program	11-6
11.3		A Sample Procedure and Trace Listing	11-8
		Procedure Nesting Level	11-8
		Time Stamp	11-10
		Statement Line Number and Text	11-10
		Statement Data	11-10

CONTENTS (continued)

PART II REFERENCE

CHAPTER 12 REFERENCE SECTION

12.1	Conventions Used in this Reference	12-1
12.2	ASSIGN	12-3
12.3	DECLARE	12-6
12.4	DESTROY	12-9
12.5	DISMOUNT	12-10
12.6	EXTRACT	12-12
12.7	GOTO	12-15
12.8	IF	12-16
12.9	LOGOFF	12-19
12.10	MESSAGE	12-20
12.11	MOUNT	12-23
12.12	OPTIONS	12-25
12.13	PRINT	12-26
12.14	PROCEDURE	12-28
12.15	PROMPT	12-29
12.16	PROTECT	12-33
12.17	RENAME	12-36
12.18	RETURN	12-38
12.19	RUN	12-39
12.20	SCRATCH	12-43
12.21	SET	12-45
12.22	SUBMIT	12-47
12.23	TRACE	12-51
12.24	USING	12-54

PART III APPENDICES

APPENDIX A	Return Code Values for VS System Utilities & Procedure language Statements	A-1
APPENDIX B	Incompatibilities between version 2.7.7 and the current version of the Procedure Interpreter	B-1
APPENDIX C	Glossary	C-1
INDEX		Index-1

FIGURES

Figure 1-1	The Options Screen	1-3
Figure 3-1	Sample GETPARM from Program COPY	3-6
Figure 10-1	The Special Menu Screen from the EDITOR	10-3
Figure 10-2	Sample Error File	10-4
Figure 10-3	The PROC Options Screen	10-5
Figure 11-1	Set Command Screen	11-4
Figure 11-2	Tracing Definition Screen	11-5
Figure 11-3	The PROC Options Screen	11-6
Figure 11-4	The TRACE Screen	11-7
Figure 11-5	TRACE Listing	11-9

TABLES

Table 5-1	SET Keywords and Their Meanings	5-1
Table 5-2	Keywords Used With EXTRACT	5-3

PART I

User Guide

CHAPTER 1

ELEMENTS OF THE LANGUAGE

VS Procedure language is used to write routines that can run system functions and supply parameter information. These routines are called procedures. Procedures are essential for running background jobs on the VS and very useful for controlling the user environment in an interactive application. This chapter describes what procedures can do, and when procedures may be used.

1.1 PROCEDURES

Procedures are much like programs. They are written in the VS Editor and stored in files. While the other languages on the VS are compiled, VS Procedure language is an interpreted language. When you run a procedure, the Procedure Interpreter is automatically invoked by the operating system. The Procedure Interpreter analyzes and executes the procedure statements one at a time, as they are encountered.

For background processing, procedures must be used to run programs and supply parameters such as file and library names, print class, maximum cpu time, and so forth. For interactive applications, this capability can be used to structure the user environment. The procedure can provide application-specific parameters. This can reduce the number of input screens the user has to respond to, or provide the user with default values for specific options on the screen.

In general, if an application repeatedly involves the same operation or sequence of operations, you can use a procedure to automate the sequence of operations. The procedure writer writes the procedure; the procedure user runs the procedure.

You can write procedures to

- Run a program, procedure, or series of programs and/or procedures;
- Set default values;
- Supply requested parameter information directly to a running program, thus modifying or eliminating screens usually displayed at the workstation during interactive processing;
- Extract information from the system and store it in variables defined in the procedure;

- Mount and dismount volumes;
- Scratch, protect, or rename files;
- Print or submit jobs;
- Perform workstation IO.

As a procedure writer, you can determine which parameter information requests are displayed, and which are entered. The procedure user can modify displayed requests. When parameter information is entered, the procedure supplies the parameter values, and bypasses the user. As a result, the user sees only the parameters that need modification, and thus can work more efficiently. Furthermore, the procedure writer can increase security by restricting the procedure user from making file assignments or specifying other types of parameters at runtime.

Example

```

Procedure for running the COBOL compiler
RUN COBOL
ENTER OPTIONS SOURCE=YES
DISPLAY INPUT LIBRARY=DRWSRCE, VOLUME=LANG

```

This procedure will run the COBOL compiler. The compiler OPTIONS screen will not be displayed and the source listing option will be YES, overriding the default of NO. The request for specification of the input source file will be displayed; however, the value for library and volume will be displayed as DRWSRCE and LANG, respectively.

Example

```

Procedure for word processing personnel
Run WP
Logoff

```

Persons who run this procedure are restricted to work within word processing. They do not have access to any of the VS data processing facilities because, after returning from word processing, the person is automatically logged off from the system.

Procedures can be used to modify the default values for system-related parameters. A procedure can be specified to be run upon user logon to the system.

Example

PROCEDURE LOGON

```

SET INLIB   = KDKSRCE,   INVOL    = LANG,
  OUTLIB    = KDKOBJ,    OUTVOL   = LANG,
  RUNLIB    = KDKPROC,   RUNVOL   = VOL444,
  SPOOLIB   = #KDKPRT,   SPOOLVOL = LANG,
                                WORKVOL = SYSTEM

```

This procedure sets the defaults for the user's input library and volume, output library and volume, run library and volume, spool library and volume, and for the work volume. Without a procedure, users would have to type in these defaults each time they log on.

Procedures can be run in the background. Background processing can be defined as running programs or procedures requiring no user intervention by submitting them to the operating system Procedure Queue, thus freeing the workstation for interactive use.

1.2 ENTERING AND RUNNING PROCEDURES

You can create, modify and run procedures in two ways: from the PROC program, or from the EDITOR. The following subsections discuss these two methods.

1.2.1 Entering and Running a Procedure from the PROC program

When you run PROC, you can run, edit, trace, check, or display a procedure. You can also run, edit, and display programs.

To run the PROC program, press PF key 1 from the command processor screen. Specify PROC for name of program, and press ENTER. Since PROC is a system program, you can leave the library and volume locations blank. The Options screen, shown in Figure 1.1, appears.

```
Wang VS Procedure Interpreter -- Version 03.00.03          Screen: OPTIONS
-----
Enter procedure name and select:
File ***** in library ***** on volume *****

(1) Run      - Run procedure or program
(2) Edit     - Edit file

(9) Trace   - Run procedure and trace execution
(10) Check  - Check procedure for correct syntax

(14) Display - Display file
(16) Exit   - End processing
```

Figure 1-1. The Options Screen

To run a program or procedure, enter the name of the procedure or program file, along with its library and volume. Then press PF key 1.

You press PF key 2 to access the EDITOR to create or edit a procedure. If you wish to create a file, do not enter any file, library or volume parameters. Simply press PF key 2. To edit a procedure, press PF key 2 after entering the file location data. Refer to the manual, VS Program Development Tools for more information about running the editor.

To run a procedure and trace the execution, press PF key 9. (Chapter 11 discusses the tracing facility.)

Pressing PF key 10 performs a syntax check on the procedure specified. (Refer to Chapter 10 for a discussion of syntax checking.)

If you wish to display the procedure code, press PF key 14. You can then run DISPLAY using the specified file, library and volume as defaults.

1.2.2 Entering and Running a Procedure from the Editor

You can create, modify, and run procedures within the EDITOR utility, as you do with programs. To create or edit a procedure, run the EDITOR and specify PROCEDURE as the language name. The procedure text, like source program text, is stored as a named file. From the Editor Special menu, you can also trace or check a procedure. (For more information about the EDITOR, refer to VS Program Development Tools.)

Once you create a source file, you can execute it directly. The Procedure Language is an interpreted language and therefore, procedures are not compiled into program files. Unlike programs, procedures are executed interpretively. As the Procedure Interpreter runs each program specified in a procedure, the user enters any parameters the procedure does not supply. If an error occurs, a message describing the nature of the problem appears on the workstation. You can edit and rerun the procedure to correct the error.

1.3 PROCEDURE LANGUAGE SYNTAX

Procedure language syntax is generally free form. However, the following rules must be observed:

- Each procedure must contain a line starting with the PROCEDURE statement. The abbreviation PROC can be used. Everything following the letters PROC or PROCEDURE on the same line is interpreted as a comment. The PROCEDURE line must precede any procedure statement, but can itself be preceded by comment lines. In the following example, the Procedure Interpreter treats PROGA LISTS PERSONNEL as a comment.

Example

```
PROC PROGA LISTS PERSONNEL  
RUN PROGB
```

```
.  
.   
.
```

- Other comment lines within the procedure must either begin with an asterisk (*) in column 1, or be enclosed in paired square brackets. The Procedure Interpreter considers all characters within the brackets, including the brackets, as blanks. The following example illustrates the use of comments in a procedure:

Example

```
* SORTS PERSONNEL
```

```
PROCEDURE  
RUN SORT [SORT BY STATE CODE]
```

```
.  
.   
.
```

- You can nest a comment within another comment. Brackets within a quoted constant are part of that constant, and are not considered comment delimiters. The ability to nest comments allows you comment-out a portion of code that contains comments.

Example

```
Procedure to demonstrate nested comments  
RUN A  
[Temporarily comment-out the code that runs B and C  
RUN B [This is a comment]  
RUN C [This is another comment]  
]  
RUN D
```

- Each procedure is made up of statements. Each statement must include a Procedure Language verb. It can also contain keywords, operators and operands, and labels.
- You must use a colon (:) to separate a statement label from the verb that follows. Do not leave any space between the label and the colon. In the following example, note that the colon immediately follows the label, without an intervening space.

Example

```
PROCEDURE  
STEP1:  RUN SORT
```

```
.  
.   
.
```

- You can use spaces to separate verbs and parts of operands in a statement. The Procedure Interpreter ignores multiple spaces between the elements of a procedure statement. Blank lines are allowed between statements or comments.
- If the syntax calls for commas, you must include them. Spaces before and after commas are optional.
- You can use uppercase and lowercase text (A-Z, a-z) within a procedure. The Procedure Interpreter shifts all unquoted text to uppercase. For example, the interpreter reads abc as ABC. However, it reads "abc" as abc.
- You can extend a procedure statement onto more than one line without special continuation characters.

Example

```
SET INLIB=MS, INVOL=VOL444, OUTLIB=  
MSCOPY, PROGLIB=MSCOPY
```

is read by the Procedure Interpreter in the same manner as:

```
SET INLIB=MS, INVOL=VOL444, OUTLIB=MSCOPY, PROGLIB=MSCOPY
```

- Procedure statements must be located between columns 1 and 71. The interpreter ignores all entries in column 72 and beyond.

WARNING

The character in column 71 of line n is considered adjacent to the character in column 1 of line n+1. The use of this feature is discouraged since if the 'B' in the following example is in column 71 and the 'RUN Y' begins in column 1, the Procedure Interpreter executes:

```
RUN X..., KEY= AB  
RUN Y
```

as

```
RUN X..., KEY= ABRUN Y...
```

- A procedure file can contain no more than 32767 records.

There is no limit on the number of labels or variables that can be defined in a procedure.

CHAPTER 2

PROCEDURE TERMINOLOGY

You construct procedures by using a series of procedure statements. Each procedure statement consists of one or more verbs, expressions, and labels. This chapter defines each of these terms in detail.

2.1 VERBS

A verb describes the operation to be performed. You must begin each procedure statement with a Procedure language verb, which may be preceded by a label, if desired. The statement verbs fall into four categories.

1. The first group of verbs can invoke system functions. These verbs include RUN, SET, SCRATCH, RENAME, PROTECT, PRINT, SUBMIT, MOUNT, DISMOUNT, LOGOFF, and EXTRACT. Each of these verbs, except for EXTRACT, performs essentially the same operation as the corresponding command and its options when issued interactively through the Command Processor. EXTRACT cannot be issued interactively through the Command Processor.

Each system function verb performs a different task. RUN runs a program or another procedure. The SET verb sets default values for file assignments, procedure submittal defaults, and print mode options. SCRATCH scratches a disk file or library. The RENAME verb renames a disk file or library. PROTECT protects a disk file or library. The PRINT verb places a print file in the PRINT Queue. SUBMIT submits a procedure or program file into the PROCEDURE queue for noninteractive execution. The MOUNT verb logically mounts a disk or tape volume. DISMOUNT logically dismounts a disk or tape volume. LOGOFF terminates procedure execution and logs the user off the system. EXTRACT extracts information from the system and stores it in the variables defined in the procedure.

2. A second group of verbs performs testing and branching within a procedure. These verbs are IF, GOTO, and RETURN. IF controls conditional execution of procedure statements. GOTO specifies the next procedure language statement to be executed by performing either a forward or backward branch. RETURN terminates procedure execution.

3. The third group of verbs performs workstation I/O. It includes MESSAGE and PROMPT. MESSAGE displays text on the workstation and continues execution of the procedure. PROMPT displays text and modifiable fields, and allows the user to respond.
4. The fourth group performs miscellaneous functions. It includes DECLARE, ASSIGN, USING, DESTROY, TRACE, and OPTIONS. DECLARE defines variables to be used within the procedure. ASSIGN assigns values to variables defined in the procedure. USING declares the formal parameters to a procedure. DESTROY closes a procedure file, attempts to scratch the procedure file, and to unlink the Procedure Interpreter. TRACE initiates or terminates tracing within a procedure. OPTIONS allows you to specify various Procedure Interpreter options.

The following alphabetized list summarizes the function of each Procedure language verb:

<u>Verb</u>	<u>Function</u>
ASSIGN	Assigns values to variables defined in the procedure.
DECLARE	Defines the variables to be used within the procedure.
DESTROY	Closes a procedure file, attempts to scratch that file, and to unlink the Procedure Interpreter.
DISMOUNT	Logically dismounts a disk or tape volume.
EXTRACT	Extracts information from the system and stores it in the variables defined in the procedure.
GOTO	Specifies the next executable procedure language statement, and performs a branch to the labeled statement specified.
IF	Controls conditional execution of procedure statements.
LOGOFF	Terminates procedure execution, closes all files and logs the user off the system.
MESSAGE	Displays text on the workstation and continues execution of the procedure.
MOUNT	Logically mounts a disk or tape volume.
OPTIONS	Allows you to specify various Procedure Interpreter options.
PRINT	Places a print file in the PRINT Queue.
PROMPT	Displays variables and constants on the workstation and also allows the user to accept data for variables.

(continued)

<u>Verb</u>	<u>Function</u>
PROTECT	Protects a disk file or library.
RENAME	Renames a disk file or library.
RETURN	Terminates procedure execution.
RUN	Runs a program or another procedure.
SCRATCH	Scratches a disk file or library.
SET	Sets default values for file assignments, procedure submittal defaults, and print mode options.
SUBMIT	Submits a procedure file into the PROGRAM or PROCEDURE Queue for noninteractive execution.
TRACE	Initiates or terminates tracing within a procedure.
USING	Declares the formal parameters to a procedure.

2.2 EXPRESSIONS

An expression consists of operators and operands. An operand can be a constant, variable, built-in function, label reference, backward reference or another expression.

There are three types of expressions: string, integer, and boolean. A string expression is any expression that evaluates to a string. An integer expression evaluates to an integer, and a boolean expression evaluates to one or zero.

You can use any type of expression anywhere a user-defined value is required, as long as the result of the expression can be converted to the type of the value required. The following subsections discuss the components of an expression.

2.2.1 Constants

A constant is a sequence of characters that represents a particular value. The two types of constants are string and integer. A string-constant is a sequence of 0 to 256 characters enclosed in paired quotes. Either single quotes (') or double quotes (") can be used. If you want to use a quote within a string, supply two occurrences of the quote.

Examples

Valid

"He said, ""I don't care.""
'He said, "I don't care."

Invalid

"He said, "I don't care."
"He said, "I don't care.""

You need to surround string-constants with quotes only when you use certain verbs. These exceptions are noted in the description of these verbs. To simplify procedure writing, the Procedure Interpreter allows quotes around all string-constants.

You can form string-constants by using the concatenation operator (!!). Concatenation of two operands takes the value of the first operand and immediately follows it with the value of the second operand.

An integer-constant is a sequence of one or more digits with an optional leading sign, and a value between -2147483647 and 2147483647.

2.2.2 Variables

A variable stores information. For example, a variable can store the name of a file. Variables are defined by using the DECLARE statement.

There are two types of variables: string and integer. String-variables can store fixed length data items that are between 0 and 256 characters long. Integer-variables can store fullword signed integers with values in the range of -2147483647 to 2147483647.

A variable is identified by a 2 to 31 character name. A variable name must start with an ampersand (&). The other characters in the name can be any combination of upper or lowercase letters, the digits 0 to 9, and the characters @, \$, #, or __. Lowercase letters are not distinguished from uppercase letters in variable names.

Examples

```
&counter
&INPUT_FILE
&i
```

When a variable is used as a procedure statement operand, the content of the variable is the value of the operand.

A global variable is declared in one procedure, but can be referenced in both that procedure and in procedures run directly or indirectly from that procedure. Once the procedure in which the variable is declared terminates, you can no longer use the variable.

If you do not explicitly declare a variable as being global, it is considered to be local. A local variable can be referenced only in the procedure in which it is declared.

When you reference a variable, the Procedure Interpreter first looks for a local declaration of that variable, and uses it if found. If no local declaration exists, the Procedure Interpreter looks for a global declaration of the variable, and uses it if one exists. If neither a local nor a global variable exists, an error occurs.

2.2.3 Substrings

A substring is a reference to a portion of a string-variable. A substring is defined by a start position and a length. The start position identifies the first character to be referenced. The length indicates how many characters, beginning with the starting character, are to be referenced. A substring can be used anywhere that a variable can be used, except as the variable being defined in the DECLARE and USING statements. The syntax for a substring is as follows:

string-variable (start[, length])

Start and length must be integer expressions. You can specify the length in two ways; with an integer expression, or with an asterisk (*). You can also leave the length unspecified. If you specify the length with an integer expression, the value of the expression defines the substring length. If you specify an asterisk, the substring length is defined to be the number of characters from the specified start to the last non-blank character in the string. When you specify neither a length nor an asterisk, the length is defined to be the number of characters from the specified start to the end of the string.

In the following examples, &USERID is a three character variable with an initial value of GS . &VAR8 is an eight character variable. &FILE is an eight character variable with an initial value of TESTFILE.

Examples

<u>Substring</u>	<u>Result</u>
ASSIGN &VAR8=&FILE(4)	"TFILE "
ASSIGN &VAR8=&FILE(4,2)	"TF "
ASSIGN &VAR8=&USERID(1) !! 123	"GS 123 "
ASSIGN &VAR8=&USERID(1,*)!! 123	"GS123 "

The first character position of a string-variable is 1, counting from left to right. The defined substring must be fully contained within the string-variable.

2.2.4 Built-in Functions

The Procedure Interpreter provides three built-in functions: &length, &time, and &date.

The built-in function, &length, takes a single string-variable or a substring as a parameter. When a string-variable is used as a parameter, the length function returns the declared length of the variable. When a substring is used as a parameter, the length function returns the length of the substring.

When you reference the function, &time, it returns a six-character string that represents the time of day. The string is in the form of HHMMSS, where HH represents hours, MM represents minutes, and SS represents seconds.

The function, &date, returns a six-character string to represent the system date. This string is in the form of MMDDYY, where MM represents months, DD represents days, and YY represents the year.

Section 6.2 describes the syntax information for built-in functions.

2.2.5 Backward References

Backward referencing occurs when you reference a statement from another statement to obtain parameter information. Refer to Section 3.4.1 for further discussion of backward referencing.

2.2.6 Operators

The following table shows the operators allowed in Procedure language:

<u>Operator Type</u>	<u>Procedure Interpreter Symbol</u>
binary arithmetic	+ - * /
unary arithmetic	+ -
relational	> >= < <= = <> EQ NEQ LT NE NLT GT NGT LE GE
logical	NOT AND OR
concatenation	!!

The order in which an expression is evaluated is determined by parentheses and by the priority of its operators. The priority of operators is shown in the following list:

Priority of Operators from Highest to Lowest Priority

unary +, unary -, NOT
* /
+ -
!!
all relational operators
AND
OR

Integer division is performed by truncating digits to the right of the decimal point. For example, 7/2 evaluates to 3.

Operands within an expression need not be of the same type. As the expression is evaluated, conversion is performed automatically, provided that conversion is possible. Both operands are converted (if necessary) to the type of the operator.

You can use boolean values with the Procedure Interpreter. The boolean values are false and true, and are represented by 0 and not 0, respectively. Boolean expressions and conversions always result in a result of either 0 or 1. Any integer value and any string value that represents an integer can be converted to a boolean value.

You cannot use the unary operator, NOT, in conjunction with a label reference because it is not always possible to determine if NOT is an operator or a label. For example, you cannot code NOT LABEL1.

Examples

<u>Boolean Expression</u>	<u>Result</u>
1 AND 1	1
0 AND 1	0
0 OR 2	1

You can always convert an integer value to a string value. If a string value consists of only integers and an optional leading sign, it can be converted to an integer value.

The types of the operands of a relational expression must either match or be compatible. Integer and boolean are compatible types. Incompatible types are integer and string, and boolean and string.

A null string is supported as a constant, "", or as the value of a variable. In comparisons to null strings, the null value is padded (if necessary) on the right with blanks to the length of the compared value. The following list contains examples of valid expressions.

<u>Expression</u>	<u>Equivalent to</u>	<u>Result</u>
1+2*3	1+(2*3)	7
1+2!!"34"	(1+2)!!"34"	334
1+(2!!"34")	1+234	235
(1=3/2) AND NOT 0	(1=1) AND 1	1

2.3 LABELS

You can identify any statement in a procedure with a label. Labels can exceed eight characters in length, but the Procedure Interpreter only examines the first eight characters. Duplicate label names, or labels containing the same first eight characters are permitted.

A label that identifies a statement must be immediately followed by a colon (:). This colon is part of the label definition, but you do not use it when you refer to a label from another statement. The first character of a label name must be alphabetic. Succeeding characters can be alphanumeric; embedded blanks are not allowed.

Labels have the following uses:

- You can reference a labeled statement from other statements in the procedure in order to obtain parameter information (backward reference).
- You can branch to a labeled statement with a GOTO statement located elsewhere in the procedure.
- You can save, inspect or test return codes by using a label. Labels for certain statements generate a return code for this purpose.

Refer to Section 3.4 for more information about labels.

CHAPTER 3

RUNNING PROGRAMS WITH A PROCEDURE (RUN, DISPLAY AND ENTER)

The most significant and useful feature of procedures is that they can be used to run programs or other procedures, and supply parameter information to those programs or procedures. (For the rest of this chapter, when the word "program" is used, procedures are also implied.)

You can run a single program or series of programs by a procedure with a minimum of user intervention. The RUN statement is used for this purpose. Like a RUN command issued from the Command Processor Menu, a RUN statement must specify the name of the program to be run, and optionally, its library and volume.

Example

```
RUN PROG1 IN GSLIB ON VOL555
```

This statement runs the program named PROG1, which is located in library GSLIB on volume VOL555. The file, library, and volume parameters are specified in a given order, separated by the connectives IN and ON.

You can write procedures that automatically run a number of programs in sequence. Individual programs are run in the order in which their corresponding RUN statements appear in the procedure. Generally, each program must complete execution before the next program is run. The following example illustrates sequential RUN statements:

Example

```
PROCEDURE RUNNING MULTIPLE PROGRAMS
```

```
STEP1:    RUN COPY
```

```
.
```

```
.
```

```
.
```

```
STEP2:    RUN DISPLAY
```

The first line identifies the file as a procedure. The comment, which is ignored by the Procedure Interpreter, identifies the procedure. STEP1 is the label of the first RUN statement, which runs the system program COPY. The second RUN statement is labeled STEP2 and runs the program DISPLAY.

The RUN statement is a powerful statement. You can perform a variety of functions by using one or more of the optional RUN clauses. This chapter discusses the Run statement's ENTER and DISPLAY clauses, and related concepts. For an overview of the RUN statement, and for information about the USING, ERROR, or CANCEL clauses, refer to Chapter 7.

3.1 LEVELS OF DEFAULT

You can specify the location of the program to be run in full or in part. When the library or volume is not specified, PROGLIB, PROGVOL, SYSLIB, and SYSVOL are used as defaults.

You can use the SET statement to specify PROGLIB and PROGVOL. If you do not specify PROGLIB and PROGVOL, their values are the library and volume in which the procedure itself is located. Refer to Section 5.1 for more information about the SET statement.

SYSLIB and SYSVOL are always the system library and system volume. You can obtain the values of PROGLIB and PROGVOL, and SYSLIB and SYSVOL by using the EXTRACT statement. Refer to Section 6.1 for more information.

You can make the following specifications:

- 1) The file, library, and volume
- 2) The file and library
- 3) The file and volume
- 4) The file alone

3.1.1 Specifying the File, Library and Volume

When you specify a file, library and volume, the system searches the library and volume for the file. If the library does not exist on the volume, you must respecify. If the library exists on the volume, but the file is not found within it, the system searches SYSLIB on SYSVOL. If the file is not found there, you must respecify.

Example

```
RUN FILEY IN LIB2 ON VOL6
```

3.1.2 Specifying the File and Library

If you do not specify a volume, the system searches for the file in the specified library on PROGVOL. If the file is not found, the system searches for the file in the specified library on SYSVOL. If the specified library does not exist on SYSVOL, you must respecify. If the specified library exists on SYSVOL, but the file is not found within it, the system searches SYSLIB on SYSVOL. If the file is not found there, you must respecify.

Example

RUN PROC1 IN LIB1

3.1.3 Specifying the File and Volume

If you do not specify a library, the system searches for the file in PROGLIB on the specified volume. If the file is not found, the system searches for the file in SYSLIB on the specified volume. If SYSLIB does not exist on the specified volume, you must respecify. If SYSLIB exists on the specified volume, but the file is not found within it, the system searches SYSLIB on SYSVOL. If the file is not found there, you must respecify.

Example

RUN FILEX ON VOL4

3.1.4 Specifying a File Name Only

If you do not specify a library or volume, the system searches for the file in PROGLIB on PROG VOL. If the file is not found, the system searches SYSLIB on SYSVOL. If the file is not found there, you must respecify.

Example

RUN COPY

When this procedure is run, the system COPY utility runs unless you have a program named "COPY" in PROGLIB on PROG VOL. In that case, your program COPY will run.

3.2 THE GETPARM REQUEST

A GETPARM request displays a formatted prompt on the workstation. This screen solicits, receives and verifies specific parameter information. You can satisfy a GETPARM either by specifying parameters at the workstation at run-time or in a procedure that includes the DISPLAY and ENTER clauses (Section 3.3). You identify each GETPARM by its parameter reference name (prname). Each parameter of a GETPARM has a keyword associated with it. These terms are discussed in this section.

3.2.1 GETPARMS

A GETPARM is a request for parameter information issued by a program. When a procedure runs a program, and that program issues a GETPARM, the prompt defined by that GETPARM is displayed on the workstation to solicit information. You can satisfy a GETPARM in one of three ways: by responding to a screen transaction at run-time, by using the ENTER clause to accept program defaults, or by using a DISPLAY clause to request the information.

Wherever possible, all VS system utilities use GETPARM requests to solicit parameter information. Therefore, you can generally provide most or all parameters that a system program requires by using a procedure.

There is a class of GETPARMS for which the system automatically provides defaults. These default GETPARMS typically request information, such as file assignment parameters, for work or print files. Since the system automatically provides default values, these GETPARM requests are never displayed at the workstation. Defaulted GETPARM requests can, however, be assigned values other than those provided by the system with a procedure. You can write a procedure to control the assignment of work files and print files, which the user cannot control when a program is run from the workstation.

3.2.2 Parameter Reference Name

Every GETPARM request is identified by a parameter reference name (prname). When a program issues a GETPARM request, that request produces a single prompt soliciting one or more parameters. The prnames associated with each GETPARM are usually unique within a program.

For easy reference, certain conventions are observed when prnames are assigned in system and file management utilities. For example, whenever a GETPARM requests parameter information for an input file, the prname for its parameter list is INPUT. Similarly, for an output file, the prname is OUTPUT. Most GETPARM requests issued by system and file management programs and associated prnames are listed in the manuals that document those programs. Refer to the VS System Utilities Reference, VS System Management Guide, and VS File Management Utilities Reference.

3.2.3 Keywords

Within a GETPARM, individual parameters are identified by keywords. When a GETPARM prompt is displayed at the workstation, the modifiable fields needing parameter values are labeled with keywords that identify the requested parameters.

A keyword is one to eight characters in length, with no embedded blanks. You can find a list of the keywords used in each GETPARM request in the manual that documents that program. Refer to the VS System Utilities Reference, VS System Management Guide, and VS File Management Utilities Reference.

3.3 SUPPLYING PARAMETER VALUES WITH DISPLAY AND ENTER

DISPLAY and ENTER clauses provide run-time parameters for a program. When a program is run interactively from the workstation, the user must supply parameter information in response to GETPARMS issued by the program. Such information typically includes the identification of files used by the program, as well as the selections offered by the program at run time.

In many instances, it is advantageous to place control of parameter specifications with the procedure writer, so that the procedure user is unable to alter these parameters. Alternately, it may be desirable to provide the user with a specific set of default values that the user can accept or override. When you run a program with a procedure that contains a RUN statement, you can use the DISPLAY and ENTER clauses to supply required parameters at run time. ENTER provides the parameters required by the program with no prompt displayed at the workstation. A DISPLAY clause issues a prompt displaying the modifiable default values specified by the procedure writer.

If a program parameter request is not satisfied by a DISPLAY or ENTER clause, a screen transaction occurs, even if the program is run from a procedure.

3.3.1 The ENTER Clause

Each ENTER clause supplies parameter information for one GETPARM request. If the parameter information provided by ENTER is complete and correct, no screen transaction is generated by the corresponding GETPARM request.

The ENTER clause must specify the pname of the GETPARM, for which values are supplied. You must also identify individual values in the ENTER clause by keywords associated with parameters in the corresponding GETPARM.

For example, the following procedure runs the COPY utility without user intervention by supplying all required parameters with ENTER clauses. (The COPY utility is described in the VS System Utilities Reference, which also lists the GETPARM requests.)

Example

PROCEDURE

RUN COPY

ENTER INPUT FILE=FILEA, LIBRARY= MS, VOLUME= VOL444

ENTER OPTIONS

ENTER OUTPUT FILE=FILEB, LIBRARY= MS, VOLUME= VOL444

ENTER EOJ

In this example, four ENTER clauses are used to satisfy all four GETPARMs issued by the COPY utility. The pnames of the GETPARMs are INPUT, OUTPUT, OPTIONS, and EOJ.

If you do not specify a value for a parameter in an ENTER clause, the parameter retains any existing default value. For example, the INPUT GETPARM for the COPY utility lists five parameters: FILE, LIBRARY, VOLUME, COPY and MODE. In the previous example, the keywords COPY and MODE are not coded and therefore, these parameters retain the default values provided for them by the COPY utility.

The ENTER OPTIONS clause satisfies the OPTIONS prompt that the COPY utility displays. Since you do not specify any options in the ENTER clause, this clause accepts all default options. To change any options, use the ENTER statement to specify the appropriate keyword and the new value. For example, to specify an indexed file rather than a sequential file, with an eight byte key starting at byte one, the ENTER clause would be as follows:

Example

```
ENTER OPTIONS FILEORG=I, KEYLEN=8, KEYPOS=1
```

The ENTER EOJ clause ends the COPY program. When you run the COPY utility from the workstation, you must, in response to the EOJ GETPARM, select either PF key 1 to rerun the utility, or ENTER, or PF key 16 to end the program. If you want the program to be rerun from a procedure, the following clause would be used, where 1 represents PF key 1:

```
ENTER EOJ 1
```

In general, whenever an ENTER clause contains insufficient or incorrect information, the corresponding GETPARM generates a screen transaction requesting you to correct the information. If the prname is incorrectly spelled, the ENTER clause will not be used to satisfy the GETPARM. Figure 3-1 shows a GETPARM request for parameter information.

***MESSAGE 0001 BY COPY

INFORMATION REQUIRED BY PROGRAM COPY
TO DEFINE INPUT
ACTIVE PROGRAM IS COPY

WANG VS COPY PROGRAM - VERSION 3.02.00

PLEASE ASSIGN "INPUT" (TO BE USED AS INPUT BY THE PROGRAM)

PLEASE SPECIFY THE COPY OPTION DESIRED:
COPY = FILE*** (OPTIONS= FILE, LIBRARY, OR VOLUME)

PLEASE SPECIFY THE FILE INFORMATION NEEDED:
FILE = FILEA*** IN LIBRARY = KOKDATA* ON VOLUME = VOL444

PLEASE SPECIFY THE MODE IN WHICH TO OPEN THE FILES TO BE COPIED:
MODE = INPUT* (OPTIONS = INPUT OR SHARED)

Press PF16 to terminate COPY.

Figure 3-1. Sample GETPARM from Program COPY

In Figure 3-1, program COPY issues this GETPARM to solicit the specifications of the input file parameters. You can fill in the desired information and press the ENTER key to satisfy this GETPARM.

The following example illustrates how the same GETPARM can be satisfied noninteractively with a procedure.

Example

PROCEDURE

 RUN COPY

 ENTER INPUT FILE= FILEA, LIBRARY= KDKDATA, VOLUME= VOL444

The relationship between the GETPARM shown in Figure 3-1, and the procedure shown above is as follows:

- In a GETPARM request, the name of the program issuing the GETPARM appears on line 1, after the Message number. In Figure 3-1, the program name is COPY. In the sample procedure, COPY is the name of the program run.
- Each GETPARM request is identified with a prname. In the GETPARM request, the prname appears on the second line of the heading (line 4). In Figure 3-1, the prname is INPUT. In the sample procedure, the prname is specified within the ENTER clause.
- Many GETPARM requests for information contain one or more modifiable fields. You refer to each field by a keyword. The keyword immediately precedes the modifiable field associated with it. In Figure 3-1, the keywords are COPY, FILE, LIBRARY, VOLUME and MODE. In the sample procedure, the keywords are specified within the ENTER clause. In the sample procedure, COPY and MODE are not specified. If you do not assign a new value to a field, it retains its default value.

3.3.2 The DISPLAY Clause

The DISPLAY clause supplies default values that the user can accept or modify when the GETPARM request screen is displayed.

The DISPLAY clause has several uses. First, you can use it in situations where some parameters are to be provided and where other parameters are to be modified by the user at runtime. A file assignment, for example, might have predefined default values supplied for the library and volume, while the user must enter the file name. You can also use DISPLAY to solicit user entry of parameters. These parameters are then used repeatedly within the procedure. The backward reference technique, discussed in Section 3.4.1, makes it possible to pass the parameters of a DISPLAY or ENTER clause to subsequent DISPLAY or ENTER clauses.

Example

PROCEDURE

RUN COPY

```
DISPLAY INPUT  FILE=FILEA, LIBRARY= MS, VOLUME= VOL444
DISPLAY OUTPUT FILE=FILEB, LIBRARY= GS, VOLUME= VOL555
ENTER OPTIONS
ENTER EOJ
```

In this example, two GETPARM requests are displayed: one for the INPUT file, and one for the OUTPUT file. The user can either modify the keyword values, or accept them as they appear. The user presses ENTER after each GETPARM screen. Since the OPTIONS and EOJ GETPARMS were satisfied with ENTER clauses, no other screens will appear while COPY runs.

For each RUN statement, you can use the DISPLAY and ENTER clauses in any order to satisfy GETPARM requests. For example, GETPARMS for the COPY utility are generated in the following order: INPUT, OPTIONS, OUTPUT, EOJ. In the previous example, the GETPARMS were satisfied by the DISPLAY and ENTER clauses in the following order: INPUT, OUTPUT, OPTIONS, EOJ. However, for programs that generate more than one GETPARM with the same prname, the GETPARMS are satisfied by the corresponding DISPLAY and ENTER clauses in the order specified within the procedure.

NOTE

The DISPLAY clause cannot be used in a procedure run as a noninteractive, background job. Any use of DISPLAY in a background procedure causes job cancellation, except when DISPLAY is overridden by an ENTER clause. (Refer to Section 3.5.)

3.4 REFERENCING WITH LABELS

You can precede any Procedure Language statement with a label. This label identifies a statement and allows a statement to be referenced.

Examples

```
STEP1:      RUN COPY

LOCATE:     DISPLAY INPUT FILE=TEST1

NEXT:       ENTER OPTIONS
```

A label causes the return code generated by certain statements to be saved for user inspection or for testing in an IF statement. You can use a label as the target of a GOTO statement. Refer to Section 4.2 for information about the IF statement, and to Section 4.3 for information about the GOTO statement. Section 4.1.1 discusses return codes in detail.

You can label a DISPLAY or ENTER clause and reference it from other procedure statements for the purpose of obtaining parameter information. This practice is known as backward referencing. Section 3.4.1 discusses backward referencing.

You can use a label in an inner nested procedure as a parameter reference name by an outer nested procedure. Refer to Section 3.5 for more information about nested procedures.

3.4.1 Backward Reference

A backward reference enables a procedure statement to obtain some or all of its required parameters from a preceding, previously executed DISPLAY or ENTER clause. A backward reference is specified by enclosing in parentheses the label of a DISPLAY or ENTER clause.

Example

```
PROC
    RUN PROG1
INFO:  DISPLAY INPUT
    RUN PROG2
    ENTER INPUT (INFO)
```

In this example, input parameters required by PROG1 are first obtained from the user with a DISPLAY clause, which is labeled INFO. The same parameters supplied to PROG1 are then supplied to PROG2 by using a backward reference. No additional screen transaction is required to satisfy the GETPARM issued by PROG2.

Backward referencing reduces redundant requests for parameters and eliminates errors introduced by inconsistent user responses. Backward reference also provides a convenient form of abbreviation when you repeatedly use a fixed set of parameters within a procedure. Backward reference saves you keystrokes, minimizes the possibility of typing errors, and makes subsequent modification or correction easier, since the parameters must be changed in only one place. There is no restriction on the number of statements that can reference a labeled DISPLAY or ENTER clause.

Example

```
PROCEDURE
    RUN COPY
OLDFILE:  DISPLAY INPUT LIBRARY=GJSDemo, VOLUME=MAIN6T
    SCRATCH (OLDFILE)
```

The procedure above runs the COPY utility, allowing the user to copy a file to another location. Although GJSDemo and MAIN6T have been provided as the default library and volume of the file to be copied, the user can respecify these values at run-time. After the copy has been performed, the copied file is scratched by the procedure. The user had to type only the information required by the COPY utility.

Information for the SCRATCH statement was obtained automatically with a backward reference.

The use of backward reference simplifies the process of correcting incorrect parameters by enabling you, or the user, to avoid multiple transactions when the same erroneous parameters are used in several places. For example, if a procedure is unable to locate the information required to locate or create a disk file, it prompts the user to supply the necessary corrections. These corrections may involve finding a file that has been moved, mounting a new volume, or finding a volume with sufficient space to create a new file. When you make a backward reference to obtain file parameters from a labeled DISPLAY or ENTER statement, the values obtained are the values actually used by the earlier procedure step, not necessarily those that you or the user specified in the referenced statement itself.

3.4.2 Backward Reference to Individual Parameters

Section 3.4.1 discussed how to obtain an entire parameter list from a previous DISPLAY or ENTER clause. A backward reference can also be used to obtain only selected parameters. Backward reference to a specific parameter is performed by using a partial backward reference.

Examples

(INPUT.LIBRARY)

(next.file)

(LOOP.DEVICE)

You create a partial backward reference by using the label of a DISPLAY or ENTER clause. This label is followed by a period, and then by the keyword that identifies the desired parameter. The partial backward reference is always enclosed in parentheses.

Example

```
procedure to copy a file to a diskette named FLOPPY
  run copy
input:  display input library=gjsdemo, volume=main6t
        enter options
        enter output file=(input.file), library=(input.library),
                           volume=floppy
        enter eof
```

The procedure above runs the COPY utility. The user is allowed to enter the name and location of the file to be copied, but is not allowed to specify the name and location of the output file. The output file and library will be the same as the input file and library because these values were obtained from the input GETPARM with partial backward references. The name of the output volume will always be FLOPPY.

3.4.3 Multiple Labels

Multiple labels can precede a statement.

Example

```
.  
. .  
CASE1: RUN PROCA  
        RETURN  
CASE2:  
CASE3: RUN PROCB  
        RETURN  
. .  
.
```

In this example, PROCB is run if a GOTO statement specifies either CASE2 or CASE3 as the label of the next statement to execute.

You cannot use multiple labels on DISPLAY and ENTER clauses.

3.5 NESTED PROCEDURES - OVERRIDING WITH DISPLAY AND ENTER

You can use the RUN statement to execute other procedures, as well as programs. The technique of executing one or more procedures from another procedure is called nesting procedures. A procedure that executes another procedure in this manner is the outer procedure. Procedures run by an outer procedure are inner procedures. An inner procedure can, in turn, run a program or another procedure. The maximum number of nesting levels depends on how your VS system is configured, but is generally 16 levels.

You can pass information from an outer procedure to an inner procedure by using the DISPLAY and ENTER clauses. You can also override the DISPLAY and ENTER clauses of an inner procedure from an outer procedure. When you nest procedures, you can modify DISPLAY and ENTER clauses of an inner procedure without permanently changing them.

Example

```
Proc Outer  
    Run INNER  
    Display OPTS  
  
Proc Inner  
    Run COPY  
OPTS:  Enter Options
```

If you run procedure INNER directly, it runs the COPY utility, and suppresses the Options screen. If you run procedure OUTER, it runs INNER, which, in turn, runs the COPY utility, and displays the Options screen. This happens because the DISPLAY clause in OUTER overrides the ENTER clause in INNER.

The label of a DISPLAY or ENTER clause in an inner procedure is used as a prname in the outer procedure. The label serves as a link between the two procedures.

In addition to overriding an ENTER with a DISPLAY (or a DISPLAY with an ENTER), you can override keyword values.

Example

```
Proc Outer
    Run INNER
    Display Infile file = xyz
```

```
Proc Inner
    Run COPY
Infile: Display Input file= TEST, library = @SYSTEM@
    Enter Options
```

If you run procedure INNER directly, it runs the COPY utility, and displays the Input screen. The filename is TEST and the library is @SYSTEM@. If you run procedure OUTER, it runs INNER, which, in turn, runs the COPY utility, and displays the Input screen. The filename is XYZ. Since the outer procedure did not specify a new library name, it remains @SYSTEM@.

When you nest procedures to more than one level, an outer procedure can override only the inner procedure that it directly runs. If you want to use an outer procedure to modify a procedure run from an inner procedure, you must use dummy DISPLAY and ENTER clauses in the inner procedure to serve as a link between the outer procedure and the innermost procedure.

Example

```
PROCEDURE OUTER
    RUN MIDDLE
OUT:  ENTER MID FILE = PAYROLL, LIBRARY= KDKLIB

PROCEDURE MIDDLE
    RUN INNER
MID:  ENTER IN

PROCEDURE INNER
    RUN COBOL
IN:   ENTER INPUT FILE = EXPENSE, LIBRARY= GSLIB
```

In this example, the ENTER clause in procedure OUTER cannot directly override values in INNER because procedure INNER is not run directly by OUTER. The ENTER clause in procedure MIDDLE is a dummy clause. It specifies no values of its own, but passes values from OUTER to INNER.

In summary, the following rules apply when using nested procedures:

- You can pass information from an outer procedure to an inner procedure by using the DISPLAY and ENTER clauses. To do so, you must label the DISPLAY and ENTER clauses of the inner procedure. You use the labels as prnames in the outer procedure.
- The specifications in the outer procedure override those in the inner procedure.
- When you nest procedures to more than one level, an outer procedure can override only the inner procedure that it directly runs.
- If you use a backward reference in an inner procedure to reference values which have been overridden by an outer procedure, the values obtained by the backward reference are those from the outer procedure.

CHAPTER 4

FLOW OF CONTROL WITHIN A PROCEDURE

Procedures are executed sequentially unless a statement directs control to another portion of the procedure. This chapter discusses methods of altering the flow of control.

You can use certain statements to terminate a procedure. Section 4.1 discusses two methods of procedure termination.

The IF statement can test a condition and cause an action as a result of the test. The IF...EXISTS statement can determine the existence of a file, and act upon that determination. Refer to Section 4.2 for more information about the IF statement. Section 4.1.1 discusses return codes in detail.

The GOTO statement specifies which statement will be executed next. Section 4.3 discusses this statement.

4.1 TERMINATING PROCEDURE EXECUTION

There are four ways to terminate a procedure.

- 1) Allow the procedure to run to the end of the file.
- 2) Use the RETURN Statement.
- 3) Use the LOGOFF statement.
- 4) Use the DESTROY Statement.

The RETURN statement is discussed in Section 4.1.1 and the DESTROY statement in Section 4.1.2. The LOGOFF statement is discussed in Section 5.9.

4.1.1 The RETURN Statement

The RETURN statement unconditionally terminates procedure execution. If the procedure was run from a program or procedure, control is returned to that program or procedure.

Certain Procedure language statements and most VS utilities generate return codes. Return codes indicate completion status. In many instances, return codes indicate the type of failure that occurred when a Procedure language statement or a program was executed.

The meaning associated with the return code for each Procedure language statement is listed in Appendix A. Return codes for selected VS utilities are also listed in Appendix A. Return codes for other utilities can be found in their respective reference manuals.

For the RETURN statement, the value specified within the optional CODE clause is returned as the procedure return code. If the CODE clause is not specified, the return code is zero.

A return code is a numeric value (-2147483647 to 2147483647). It is placed in the label of a statement. Note that many VS utilities only support return codes in the range 0 to 999999.

Example

```
PROC
TEST1:  RUN FORTRAN
        RETURN CODE = TEST1
```

This procedure causes the return code associated with the RUN statement to be returned as the procedure return code. If the procedure is run interactively, from the Command Processor or from the Editor, the return code value is displayed on the screen at procedure completion.

Procedures always produce zero return codes unless the CODE clause is specified in a RETURN statement that terminates the procedure. This is true whether the procedure is run from the Command Processor, EDITOR, or another procedure. If you use the CODE clause of the RETURN statement in a procedure run from the Command Processor or EDITOR, the return code displays at the workstation upon procedure termination.

You can add an integer constant to the return code to identify the section of the procedure returning a code. You can use return codes in this manner as a debugging tool for procedures that run multiple procedures.

When multiple labels precede a statement, you can only use the label closest to the statement for return code testing or return code generation.

The return code produced by an inner procedure can be tested by an outer procedure. If an inner procedure is executed from a labeled RUN statement in an outer procedure, the return code of the inner procedure is associated with the label of the RUN statement in the outer procedure.

The following examples demonstrate use of nested procedures that use return codes.

Example

```
PROCEDURE OUTER
RTNCODE:   RUN INNER
           IF RTNCODE LE 4 GOTO END
           RUN FIXIT
END:       RETURN CODE = RTNCODE
```

```
PROCEDURE INNER
COMPILE:   RUN COBOL
           ENTER INPUT FILE = TEST
           ENTER OPTIONS
           ENTER OUTPUT FILE = TESTOBJ
           RETURN CODE = COMPILE
```

Procedure OUTER runs procedure INNER. Procedure INNER, in turn, runs the COBOL compiler. Upon completion, INNER returns the COBOL return code, which indicates the result of the compilation. This code is tested by OUTER. If it is greater than 4 (indicating one or more serious compilation errors), program FIXIT is run (the code for procedure FIXIT is not shown in this example). If the return code obtained from INNER is less than or equal to 4 (indicating no serious compilation errors), that return code is displayed on the workstation.

4.1.2 The DESTROY Statement

The DESTROY statement closes the currently executing procedure file, attempts to scratch it, and returns control to the invoking routine. It is primarily for use with program generated procedures that are intended for use only once.

4.2 CONTROLLING CONDITIONAL EXECUTION

The IF statement controls conditional execution of procedure statements. The IF statement has two forms. An IF statement tests a condition and acts upon the results of that test. An IF...EXISTS statement determines whether a file, library or volume exists, and acts upon the result of that determination. The IF statement is discussed in Section 4.2.1. The IF...EXISTS statement is discussed in Section 4.2.2.

4.2.1 The IF Statement

The general format of the IF statement is:

```
[label:]... IF relational expression [THEN] [label:]...any-statement
```

A relational expression evaluates to a boolean value of true or false. If the expression is true, the "then" portion of the IF statement is executed. If the expression is false, the "then" portion of the IF statement is ignored.

A relational operator defines the relationship between two operands. You can represent the relational operator with a symbol or with an abbreviation. Table 4-1 shows how relational operators are defined for use with Procedure language:

Table 4-1. Relational Operators and Their Meanings

Symbol	Abbreviation	Meaning
>	GT	Greater than
>=	GE, NLT	Greater than or equal to (not less than)
<	LT	Less than
<=	LE, NGT	Less than or equal to (not greater than)
=	EQ	Equal to
<>	NEQ, NE	Not equal to

You can use the relational operators to compare constants, variables, labels, and substrings, in any combination. The types of the operands in a relational expression must either match, or be compatible. Compatible types are integer and boolean. Incompatible types are integer and string, and boolean and string.

Any statement can follow the relational expression. If the relational expression is true, the statement within the THEN clause is executed. If the relational expression is false, the statement within the THEN clause is ignored, and the statement following the IF statement is executed.

Examples

```
IF LABEL1 NE 5 THEN RUN COPY
if &counter = 100 then goto end
IF &DATE = "122583" THEN PROMPT CENTER "Merry Christmas!"
```

4.2.2 The IF...Exists Statement

The IF...EXISTS statement determines the existence of a file, library, or volume.

If you specify FILE, and the specified file exists, the statement within the THEN clause is executed. Otherwise, the next statement in the procedure is executed. If you specify NOT, and the specified file does not exist, the statement within the THEN clause is executed. Otherwise, the next statement in the procedure is executed.

Example

```
IF EXISTS FILE GRAPH1 IN CCDATA ON SYSTEM THEN RUN DISPLAY  
RUN COPY
```

If the file GRAPH1 exists in the library CCDATA on volume SYSTEM, DISPLAY is run. However, if this file does not exist, COPY is run next.

If you specify LIBRARY, and the specified library exists, the statement within the THEN clause is executed. Otherwise, the next statement in the procedure is executed. If you specify NOT, and the specified library does not exist, the statement within the THEN clause is executed. Otherwise, the next statement in the procedure is executed.

If you specify VOLUME, and the specified volume exists, the statement within the THEN clause is executed. Otherwise, the next statement in the procedure is executed. If you specify NOT, and the specified volume does not exist, the statement within the THEN clause is executed. Otherwise, the next statement in the procedure is executed.

Examples

```
if exists library #SMDPRT on VOL999 then goto PART1  
IF EXISTS VOLUME LANG THEN SCRATCH KDKSRC  
IF NOT EXISTS VOLUME SYS444 THEN MOUNT SYS444 ON 35
```

A file, library, or volume does not exist if

- The specified volume is being used exclusively by another user.
- There is insufficient buffer space to perform the IF EXISTS operation.
- There is a VTOC error.
- A disk I/O error occurs.
- The specified volume is not mounted.
- The specified file and/or library cannot be found on the specified volume.

4.3 FORWARD OR BACKWARD BRANCHES: THE GOTO STATEMENT

The GOTO statement specifies the next executable procedure language statement to be executed, and performs either a forward or backward branch.

Example

```
PROCEDURE
    GOTO STEP2
    RUN COPY
STEP2:  RUN DISPLAY
```

The GOTO statement in the example above causes the RUN COPY statement to be ignored. RUN DISPLAY is the next statement to be executed.

Example

```
PROCEDURE
    DECLARE &VALUE1 INTEGER INITIAL 0
REPEAT: RUN PROG1 IN CCData ON VOL999
    ASSIGN &VALUE1 = &VALUE1 + 1
    IF &VALUE1 < 5 GOTO REPEAT
LAST:  LOGOFF
```

In this example, if &VALUE1 is less than 5 when the IF statement is executed, the RUN statement, labeled REPEAT, is executed next. If variable, &VALUE1, is greater or equal to 5, the GOTO statement is ignored and the LOGOFF statement is executed.

You can label multiple statements with the same label. The first occurrence of a label following the GOTO statement in the procedure text is the target.

If no label follows the statement, the target is the closest preceding occurrence of the label, preceding the GOTO statement in the procedure text.

Example

```
100    PROCEDURE
200        GOTO A
300    B:  RUN X
400    A:  RUN Y
500        GOTO A
600    B:  RETURN
700    A:  RUN Z
800        GOTO B
```

This procedure is executed in the following order: 200, 400, 500, 700, 800, 600.

If no label exists, an error occurs.

The backward GOTO statement works differently from a backward reference. The backward GOTO references the closest textual occurrence of a label. The backward reference finds the most recently executed occurrence of a labeled DISPLAY or ENTER clause.

CHAPTER 5

SYSTEM CALL STATEMENTS

You can use system call statements to invoke system functions. These statements support tasks involved in program and file processing. They can also be used to manipulate peripheral devices. There are eleven system call statements. This chapter discusses ten of these statements: SET, EXTRACT, SCRATCH, RENAME, PROTECT, PRINT, SUBMIT, MOUNT, DISMOUNT, and LOGOFF. Refer to Chapters 3 and 7 for a discussion of the eleventh, the RUN statement.

5.1 SETTING DEFAULT VALUES: THE SET STATEMENT

The SET statement is similar to the SET USAGE CONSTANT command that is issued interactively through the Command Processor (refer to the VS Programmer's Introduction). You can use the SET statement to specify the default names used by commands and programs to identify input, output, the program libraries and volumes, and the work and spool volumes. You can also use SET to specify job submittal defaults, a default file protection class, and print mode options.

When you invoke the SET Usage Constant command interactively from the Command Processor, you must associate a value with a keyword. Each keyword identifies a field for which you must specify a default parameter value. Table 5-1 illustrates the SET keywords and their meanings. Refer to The VS Programmer's Introduction for a list of the associated values and their meanings.

Table 5-1. SET Keywords and Their Meanings

Keyword	Meaning	Keyword	Meaning
FILECLAS	File protection class	PROGLIB	Program library
FORM#	Printer form number	PROGVOL	Program volume
INLIB	Input library	PRTCLAS	Spooled print file class
INVOL	Input volume	PRTFCLAS	Print file protection class
JOBCLASS	Background job class	RUNLIB	Run library
JOBLIMIT	Background job CPU	RUNVOL	Run volume time limit

(continued)

Table 5-1.(continued)

Keyword	Meaning	Keyword	Meaning
JOBQUEUE LINES	Background job status # lines per page	SPOOLIB SPOOLSYS	Spool library Remote spool system
OUTLIB	Output Library	SPOOLSYSRC	SPOOLSYS return code
OUTVOL	Output volume	SPOOLVOL	Spool volume
PRINTER	Output printer number	WORKVOL	Work volume
PRNTMODE	Print mode		
PRNTMODE	Print mode		

Two keywords, PROGLIB and PROGVOL, are offered through the SET statement, but are not available through the SET Usage Constants command. PROGLIB and PROGVOL are used as the default library and volume for the RUN statement (Section 3.1). If you run a program interactively from the Command Processor instead of from a procedure, the default library and volume are RUNLIB and RUNVOL.

If you do not explicitly set PROGLIB and PROGVOL with the SET statement, the library and volume in which the procedure itself is located are used as the values of PROGLIB and PROGVOL.

All parameters except PROGLIB and PROGVOL become default parameters after the procedure is terminated, and remain in effect until you log off. PROGLIB and PROGVOL remain effective only while the procedure that set them is executing.

The system defines the work library default name. You cannot modify the default name for this library with a SET statement. For this reason, WORKLIB is not listed as a legal keyword, even though it appears in the File Defaults display of the SET Usage Constants command of the Command Processor.

Example

```
SET INLIB= GS, INVOL= VOL555, PROGLIB= MS, PROGVOL= VOL444,
PRNTMODE= H
```

When this statement is executed in a procedure, GS on VOL555 becomes the default input library and VOL555 is the input volume. MS becomes the default program library with VOL444 as the volume; the print mode is set to HOLD. Subsequently, any procedure statement, program, or command that refers to the input library uses the names GS and VOL555, unless you supply different library and volume names at run-time. Similarly, all programs this procedure runs are assumed to be located in library MS on VOL444, unless otherwise specified. The parameters not defined in a SET statement are not changed. In the previous example, the output library and volume are not specified; default values for these parameters remain unmodified, unless specified in another SET statement.

SPOOLSYS allows a WANGNET user to specify a remote system as the system to which print files are automatically routed using FILE-TRANSFER.

The return codes for SPOOLSYS are as follows:

- 0 The System Name has been found in the Network Configuration.
- 4 The System Name has not been found in the Network Configuration.
- 8 The System Name could not be validated because it was not possible to obtain enough buffer space to send a message to the System Task. Try again later.
- 12 The System Name could not be validated. The System Task is not running. IPL.

Example

```
DECLARE &SPOOLSYSRC integer
SET SPOOLSYS= SYS2T, SPOOLSYSRC = &SPOOLSYSRC
```

When the SET statement is executed, an attempt is made to set SPOOLSYS to SYS2T. The return code associated with that attempt is stored in &SPOOLSYSRC.

5.2 THE EXTRACT STATEMENT

The EXTRACT statement extracts information from the system and stores it in the variables specified in the statement.

The EXTRACT statement can be executed with specified keywords. The keywords identifying fields for which data can be extracted are presented in Table 5-2.

Table 5-2. Keywords used with EXTRACT

Keyword	Meaning
CURLIB	Library where the current program resides. In general, this library will be the library containing the Procedure Interpreter.
CURVOL	Volume where current program resides. In general, this volume will be the volume containing the Procedure Interpreter.
DISKIO	Count of disk IOs for this run
FILECLAS	File protection class

(continued)

Table 5-2. (continued)

Keyword	Meaning
FORM#	Printer form #
INLIB	Input library
INVOL	Input volume
LINES	Lines-per-page for print files
OTIO	Count of IOs for other devices not included under WSIO, DISKIO, PRINTIO, or TAPEIO
OUTLIB	Output library
OUTVOL	Output volume
JOBCLASS	Background job class
JOBLIMIT	Background job limit
JOBQUEUE	Background job status
PRINTER	Printer number
PRINTIO	Count of print IOs this run
PRNTMODE	Print file output mode
PROGLIB	Program library. If you have not set PROGLIB with a SET statement, then its value is the library containing the procedure that you are currently running.
PROGVOL	Program volume. If you have not set PROGVOL with a SET statement, then its value is the volume containing the procedure that you are currently running.
PRTCLAS	Print class for print files
PRTFCLAS	Print file protection class
RUNLIB	Run library
RUNVOL	Run volume
SPOOLIB	Spool library
SPOOLSYS	System used for remote print routing
SPOOLVOL	Spool volume

(continued)

Table 5-2. (continued)

Keyword	Meaning
SYSLIB	System library
SYSVOL	System volume
SYSWORK	System work library, used for paging files, system task queues and other system functions
TAPEIO	Count of tape IOs this run
TASK#	Unique task identifier
TASKTYPE	Task type
USERID	User logon identification
USERNAME	User name derived from system user list
VERSION	Operating system version number
WORKLIB	Work library
WS	Workstation number
WSIO	Count of workstation IOs this run

Example

```
step1: EXTRACT &var = SYSVOL
```

When this statement executes, the Procedure Interpreter assigns the system volume name to the variable &var.

Example

```
EXTRACT &recs= RECORDS USED BY KDKTST IN KDKPROC ON LANG
```

The Procedure Interpreter sets the variable &recs to the number of records used by the file KDKTST, which is located in library KDKPROC on volume LANG. If the file does not exist, &recs receives the value of negative one (-1).

5.3 SCRATCHING A FILE OR LIBRARY: THE SCRATCH STATEMENT

The SCRATCH statement deletes all reference to a specified file from the disk Volume Table of Contents (VTOC) and frees the space occupied by the file for other use.

Example

SCRATCH FILE1 IN MS ON VOL444

This SCRATCH statement scratches file FILE1 in library MS on volume VOL444.

You can also use the SCRATCH statement to scratch an entire library. All files, except the files for which you do not have access rights, or that have unexpired retention periods, are deleted by a SCRATCH LIBRARY statement.

Example

SCRATCH LIBRARY MS ON VOL444

This SCRATCH statement scratches library MS from volume VOL444.

5.4 RENAMING A FILE OR LIBRARY: THE RENAME STATEMENT

The RENAME statement changes a file or library name in the disk VTOC to a new name, without altering the file or library contents.

Example

RENAME FILE1 IN MS ON VOL444 to FILE2

The name, FILE2, becomes the name of the file, and the name, FILE1, no longer accesses the file.

The RENAME statement can also be used to move a file from one library to another.

Example

RENAME FILE1 IN MS ON VOL666 to FILE1 IN KDKLIB

This RENAME statement effectively moves FILE1 file in library MS on VOL666 to library KDKLIB on VOL666.

5.5 PROTECTING A FILE OR LIBRARY: THE PROTECT STATEMENT

The PROTECT statement changes the protection parameters for a disk file or library. These parameters include the file or library owner, the file or library protection class, and the file or library retention period.

The following statement modifies the protection parameters of FILE1 so that KDK becomes the new owner of record, the file protection class becomes A, and the retention period becomes 21 days:

Example

```
PROTECT FILE1 in GSDEMO on VOL444 TO  
OWNER = KDK, PERIOD = 21, FILECLASS = A
```

You can also use the PROTECT statement to change the protection parameters of every file in a library.

Example

```
PROTECT LIBRARY GSDEMO ON VOL444 TO OWNER = KDK
```

5.6 QUEUING A PRINT FILE: THE PRINT STATEMENT

You can use the PRINT statement to queue a print file into the PRINT queue. This statement is similar to the PRINT option of the Manage FILES/LIBRARIES command issued interactively through the Command Processor. The file can either be printed as soon as a printer is available, or can be held for later use. After printing the file, you can delete it, requeue it, or save it in your print library. Additionally, you can specify the print class, form number, and number of copies.

Example

```
PRINT EDIT004 IN #KDKPRT ON SYSTEM, CLASS= A, FORM#= 000,  
COPIES= 2, DISP= SAVE
```

This statement enters the print file EDIT004 in library #KDKPRT on volume SYSTEM onto the PRINT Queue. The file class is specified as A, the form number as 000, and two copies will be printed. After printing, the file is saved in library #KDKPRT, but is not listed again in the print queue.

5.7 BACKGROUND PROCESSING: THE SUBMIT STATEMENT

Sometimes, it is desirable to run programs with no user intervention. The process of running programs without user intervention after the initial command is called background processing. You can use background processing on the VS by using procedures.

The SUBMIT statement queues a program or procedure file into the PROCEDURE Queue for execution on a noninteractive basis. A background procedure is canceled if the procedure performs workstation I/O, exceeds its cpu limit, or contains an error. When a submitted procedure is canceled, a one page "indicative dump" that resembles a DEBUG screen display is generated. The error listing is automatically spooled to the default printer, since the default PRNTMODE for background tasks is

SPOOL. You can direct the listing to a specified library and volume by coding a SET statement in the canceled procedure. The statement should set PRNTMODE to KEEP, and SPOOLSYS and SPOOLVOL to the library and volume that is to receive the error listing. Refer to Section 5.1 for more information about the SET statement.

The following statement enters the procedure file in PROC1 in library PROCLIB into the PROCEDURE Queue under the job name TEST:

Example

```
SUBMIT PROC1 IN PROCLIB AS TEST, CLASS= A, STATUS=
HOLD, DUMP= YES, CPULIMIT= 0:05:00, ACTION= WARN, DISP=
REQUEUE
```

The procedure class specified is A, and the procedure is to be held in the queue until you release it or remove it. When the procedure has been executed, the operator is sent a message if the procedure exceeds five minutes of CPU time. If the procedure is terminated abnormally, a dump listing is created. When the job has been successfully executed, it is listed again in the PROCEDURE Queue.

You can also use the SUBMIT statement to pass selected local variables to the submitted procedure or program, and all global variables to a submitted procedure. You can also pass your current environment. The submitted program or procedure then uses the same usage constants as the user's working environment.

In certain cases, the Procedure Interpreter must generate a procedure that runs your procedure. The Procedure Interpreter submits the generated procedure instead of submitting your procedure directly. The Procedure Interpreter must generate a procedure to support the following conditions:

- The USING clause is coded.
- GLOBALS = YES is specified.
- ENVIRONMENT = YES is specified.
- The file referenced by the specified file name is a program file.

If you examine the procedure queue (accessible from the Command Processor), you will notice that the generated procedure's procedure-id is the name you specified within the AS clause of the SUBMIT statement. If you did not specify the AS clause, the generated procedure's procedure-id is the name of the procedure that you submitted. The file name is always chosen by the Procedure Interpreter.

Once submitted, the generated procedure runs your procedure. After your procedure has completed, the generated procedure is automatically scratched. However, if you specify STATUS = HOLD, the generated procedure is placed on the procedure queue, and is not executed until you release it. If you remove the generated procedure from the queue without running it, it is not automatically scratched. Instead, it remains in the system work library. The name of the system work library can be obtained by extracting SYSWORK with the EXTRACT statement.

Procedures used for background processing can include any of the available procedure statements. No screen transactions can be generated during background processing, except in relation to the MOUNT and DISMOUNT statements. Any other statement that attempts to force a screen transaction results in cancellation of the procedure.

The MOUNT and DISMOUNT statements were designed for background processing applications. These statements force screen messages at the operator's console only. These messages will not cause a submitted procedure to terminate abnormally.

Since all other screen displays are prohibited during background procedures, the procedure writer must include values for all parameter requests issued by background programs.

Example

Procedure to specify a file, library and volume for COBOL program

```
declare &file, &library string(8)
declare &volume string(6)
prompt
  center 'Enter the name of a COBOL source program';;
  'Library =', upper &library,
  'Volume =', upper &volume
submit subcobol using &file, &library, &volume
```

Procedure SUBCOBOL, which compiles a COBOL program in the background

```
Using &file, &library string(8), &volume string(6)
Run COBOL
enter input file= &file, library= &library, volume= &volume
enter options
enter output file &file, library= GSOBJ, volume= LANG
```

The first procedure in this example requests the user to specify a COBOL source program. The first procedure then submits the second procedure as a background task, and passes the specified file name and its location to SUBCOBOL.

The second procedure, SUBCOBOL, receives the file, library and volume parameters. When this procedure runs COBOL, these parameters are used.

Since the DISPLAY clause results in a screen transaction, you cannot use the DISPLAY clause in background processing. The DISPLAY clause can only be used in background processing if it is located in an inner procedure, and an ENTER clause in an outer procedure overrides it. In any other case, a DISPLAY clause used in background processing terminates the procedure abnormally.

It is advisable to test each procedure in Interactive mode before you submit it to background processing. In this way, you can identify and eliminate any screen transactions that would cause abnormal termination during background processing.

5.8 LOGICALLY MOUNTING AND DISMOUNTING A VOLUME

The MOUNT and DISMOUNT statements are used to automate mounting and dismounting of disk and tape volumes by permitting these operations to be performed from procedures. These statements are analogous to the MOUNT and DISMOUNT commands issued interactively through the Command Processor.

5.8.1 The MOUNT Statement

You can use the MOUNT statement to logically mount a specified disk or tape volume on a specified unit, prior to physical volume mounting.

When you perform a mount through background processing, a display requesting the mount is forced at the highest priority Operator's Console available. (Refer to the VS System Operations Guide.) This display requests that the operator perform the physical mounting operation.

The following statement mounts the disk volume named VOL444 on unit number 11:

```
MOUNT DISK VOL444 ON 11 WITH STANDARD LABEL FOR EXCLUSIVE USAGE
```

VOL444 has a standard label and is reserved for use only at the workstation from which it is mounted. The volumes mounted as EXCLUSIVE through a background processing job can be dismounted only through that background processing job. Such mounts must be dismounted before the background job can terminate.

5.8.2 The DISMOUNT Statement

The DISMOUNT statement logically dismounts a specified disk or tape volume prior to physical volume dismounting. The following statement dismounts the tape volume named TAPEVOL:

```
DISMOUNT TAPE TAPEVOL
```

A DISMOUNT issued from a background job forces an operator message, which remains on the screen until you press ENTER from the Operator's Console. Pressing ENTER signifies that the DISMOUNT is complete.

5.9 LOGGING OFF: THE LOGOFF STATEMENT

The LOGOFF statement terminates your current session. It terminates all of your currently executing programs and procedures, closes all files, and automatically issues the Command Processor LOGOFF command.

When programs or procedures are run from the EDITOR or a menu, the LOGOFF statement returns control to the EDITOR or menu rather than canceling the user's session.

Example

Procedure to demonstrate MOUNT, DISMOUNT, and LOGOFF

First: MOUNT DISK FLOPPY ON 57 WITH NO LABEL

RUN DISKINIT

ENTER FUNCTION FUNCTION= INITIALIZE, VOLUME= FLOPPY

ENTER INPUT LABEL=NL

ENTER EOJ 16

DISMOUNT DISK FLOPPY

LOGOFF

In this example, a screen appears to tell you to mount a non-labeled disk named "FLOPPY" onto device numbered "57". Next, the DISKINIT program runs, and disk FLOPPY is initialized. After DISKINIT is completed, a screen appears asking you to dismount the disk. Finally, you are logged off of the system, or, if you are running this procedure from within the editor, you are returned to the editor.

CHAPTER 6

THE OPTIONS STATEMENT AND BUILT-IN FUNCTIONS

The `OPTIONS` statement allows specification of various Procedure Interpreter options. The built-in functions provide various types of information that might not otherwise be obtained without writing specialized procedures or programs. This chapter describes the `OPTIONS` statement and the built-in functions.

6.1 THE OPTIONS STATEMENT

The `OPTIONS` statement allows you to specify various Procedure Interpreter options. Currently, it supports only one option, the `PROCMSG` option.

Normally, the message "Procedure XXX in Progress" is displayed on the workstation when a procedure is run. It is also displayed after a `PROMPT` statement has been executed. It may be desirable to suppress this message between two successive `PROMPT` statements, or when running other procedures from a procedure.

Example

```
PROC MAIN
  OPTIONS PROCMSG = NO
  RUN SUB1
  RUN SUB2
```

When this procedure is run, the message "Procedure MAIN in progress" is displayed on the workstation. Although `SUB1` and `SUB2` are also procedures, they do not display their own "Procedure in progress" message since `PROCMSG = NO` was specified. If the `OPTIONS` statement were omitted, three different "Procedure in progress" messages would be displayed.

If a procedure specifies `PROCMSG = NO`, display of the message remains disabled until the current procedure ends or until `PROCMSG = YES` is specified. If you are nesting procedures the option is effective for the current procedure, and for any procedures run from that procedure.

An inner procedure can respecify `PROCMSG` without affecting the specification in the outer procedure. For example, Procedure A disables display of the message, and then runs Procedure B. Procedure B, in turn, enables display of the message. Display of the message remains enabled

until Procedure B returns to Procedure A. At this time, display of the message is again disabled.

6.2 BUILT-IN FUNCTIONS

The Procedure Interpreter provides three built-in functions: &length, &time, and &date. Since these functions are pre-defined, you use them without declaration.

If you declare a local variable with the same name as one of the built-in functions, you cannot use that function within the procedure where it has been declared. However, the built-in function will be accessible in a subsequent procedure, or in a procedure at a higher link level.

If you declare a global variable with the same name as one of the built-in functions, that function is not available in that procedure, or in any subsequent procedures. However, it will be available in procedures at higher link levels.

6.2.1 &length

The built-in function, &length, takes a single string-variable or a substring as a parameter. When a string-variable is used as a parameter, the length function returns the declared length of the variable. When a substring is used as a parameter, the length function returns the length of the substring.

Format

&length (parameter)

The function, &length, can be used to determine the number of non-blank characters in a string-variable.

Example

```
procedure
declare &file string(8) initial 'ABC'
declare &filename_len integer
assign &filename_len = &length(&file(1,*))
```

In the example above, variable &filename_len receives 3 because the substring &file(1,*) references only the first three characters of &file. Refer to Section 2.2.3 for more information about substrings.

6.2.2 &time

When the function, &time, is referenced, it returns a six-character string that represents the time of day. The time is in the form HHMMSS, where HH is the number of hours based on a 24 hour clock, MM is the number of minutes, and SS is the number of seconds.

Format

&time

Example

```
procedure
declare &saved_time string(6)
declare &formatted_time string(8)
assign &saved_time = &time
assign &formatted_time = &saved_time(1,2) !! ':' !! &saved_time(3,2)
                        !! ':' !! &saved_time(5,2)
```

In the example above, if the time is 8:32 and 17 seconds A.M., the value "08:32:17" is assigned to &formatted_time. If the time is 8:32 and 17 seconds P.M., the value "20:32:17" is assigned to &formatted_time.

6.2.3 &date

The function, &date, returns a six-character string that represents the system date. The date is in the form of MMDDYY, where MM represents the month number, DD represents the day of the month, and YY represents the year.

Format

&date

Example

```
procedure
declare &saved_date string(6)
declare &formatted_date string(8) initial "  /  /  "
assign &saved_date = &date
assign &formatted_date(1,2) = &saved_date(1,2)
assign &formatted_date(4,2) = &saved_date(3,2)
assign &formatted_date(7,2) = &saved_date(5,2)
```

In the example above, if the date is December 15, 1983, the final value of &formatted_date becomes 12/15/83.

CHAPTER 7

RUNNING PROGRAMS AND PROCEDURES WITH A PROCEDURE

This chapter completes the discussion about the RUN statement that was introduced in Chapter 3. Specifically, this chapter discusses in detail the USING, the ERROR EXIT IS, and the CANCEL EXIT IS clauses of the RUN statement. The USING statement is also discussed articles needed for translation in this chapter.

7.1 UNSUCCESSFUL LINKS- THE ERROR EXIT IS CLAUSE

The ERROR EXIT [IS] clause allows you to transfer control to another labeled statement if the system could not run a specified program or procedure. If you specify ERROR EXIT within a RUN statement, and the program or procedure cannot be run, the label of the RUN statement will receive a return code to indicate why the RUN failed. Refer to Appendix A for a list of return codes for the RUN statement.

Example

Procedure to test the error clause

```
R:      Run NEWCOPY in KDKPROG on LANG error exit is error1
      Return
```

```
Error1: IF R NE 20 then Return
      Rename COPYPROG in KDKPROG on LANG to NEWCOPY
      GOTO R
```

The Procedure Interpreter attempts to run the file NEWCOPY. Since this file does not exist, a return code of 20 (FILE CANNOT BE FOUND) is assigned to R. Since an error exit is specified, the next statement executed is the IF statement labeled ERROR1. The return code for R equals 20, so the RENAME statement is executed. The GOTO statement directs execution back to the RUN statement labeled "R", which runs the procedure NEWCOPY. Finally, the procedure terminates with the RETURN statement.

If you do not specify an ERROR EXIT IS clause, and an unsuccessful link occurs, a GETPARM appears. This GETPARM explains the problem and requests a different file. Figure 7.1 is an example of this GETPARM screen.

```

*** MESSAGE BY @PROC@
INFORMATION REQUIRED BY PROGRAM PROC
TO DEFINE ERROR
ACTIVE SUBPROGRAM IS @PROC@

Correction required to continue interpreting
procedure EDIT008 in #KDKWORK on LANG.

000200 R:      Run NEWCOPY in KDKPROG on LANG

File cannot be found

Respecify program or procedure to be run and press ENTER:

FILE      = NEWCOPY*      LIBRARY = KDKPROG*      VOLUME  = LANG**

```

Figure 7.1 The Error GETPARM Screen

7.2 CANCELLED PROCEDURES- THE CANCEL EXIT IS CLAUSE

The CANCEL EXIT [IS] clause allows you to transfer control when a successfully linked-to program or procedure is cancelled.

Example

Procedure to test the cancel clause

```
C: Run errortst in kdkproc on lang cancel exit is cancell
   return code = 53
```

```
cancell:      prompt center 'Cancel was taken'
              return code = 5000
```

This procedure runs ERRORTST. If a link is made to ERRORTST, and ERRORTST is cancelled while running, the next statement executed is the PROMPT statement labeled CANCEL1. After the prompt, the procedure ends, and returns a return code of 5000. If ERRORTST had not been cancelled, the return code would have been 53.

The ERROR EXIT clause and the CANCEL EXIT clause must follow the USING clause in the RUN statement. You can specify the ERROR and the CANCEL clauses in any order, but only one ERROR and one CANCEL clause per RUN statement is allowed.

If CANCEL EXIT is specified and a cancel occurs, the label of the RUN statement will not receive a return code. If the CANCEL EXIT clause is not included, the procedure running the cancelled procedure is also cancelled.

7.3 DECLARING THE FORMAL AND ACTUAL PARAMETERS

You use the USING clause of the RUN statement to specify the actual parameters to be passed to a program or procedure. You use the USING statement to declare the formal parameters of a procedure.

In Procedure Language, parameters are passed in three ways: by reference, by value, and by value-result. The following subsections describe these ways of parameter passing, and describe the USING clause of the RUN statement, and the USING statement.

7.3.1 Pass by Reference

Only variables can be passed by reference. When a variable is passed by reference, it is the address of the variable that is passed. The procedure or program that receives the parameter can modify the variable directly by using the address.

7.3.2 Pass by Value

Substrings and expressions consisting of other than a single variable are passed by value. When a parameter is passed by value, it is the address of a copy of the value that is passed. The procedure or program that receives this parameter can modify the copy, but not the actual value.

7.3.3 Pass by Value-Result

Only string variables can be passed by value-result. Passing by value-result is like passing by reference in that the address of the variable is passed to the called procedure. However, the passed variable is not actually modified until the called procedure returns. A pass by value-result occurs only when a string variable of length *n* is passed to a procedure, which declares that variable as greater than length *n*. To prevent the called procedure from writing past the end of the variable, the called procedure makes a copy of the variable, and uses this copy within the called procedure. Before returning to the calling procedure, the called procedure copies *n* characters of the copy to the actual variable that was passed.

Pass by value-result is only supported when passing parameters from a procedure to a procedure; string variables passed from a procedure to a program are always passed by reference.

In Procedure Language, you do not specify how a parameter is passed; a parameter is passed according to its type.

7.3.4 The USING Clause and the USING Statement

The USING clause is part of the RUN statement. It Allows you to pass parameters to the program or procedure to be run. If parameters are passed to a procedure, they are received by the USING statement of that procedure.

The USING statement receives parameters passed from a procedure or program.

When the USING statement is used, it must be the first statement following the PROCEDURE statement.

The USING statement defines the parameters passed to the procedure. Any program that uses appropriate types and lengths of parameters can call to a procedure and pass data.

The number of parameters passed must equal the number of parameters declared in the USING statement.

Example

Procedure CALLER, which passes parameters to CALLEE

```
Declare &parm1 integer    initial 5
Declare &parm2 string(3)  initial 'ABC'
Declare &parm3 string(8)  initial 'Hello'
Prompt 'P1 = ', &parm1; 'P2 = ', &parm2; 'P3 = ', &parm3(1,1)
Run CALLEE using &parm1, &parm2, &parm3(1,1)
Prompt 'P1 = ', &parm1; 'P2 = ', &parm2; 'P3 = ', &parm3(1,1)
```

In procedure CALLER, &parm1 is declared an integer with an initial value of 5. &parm2 is declared a three character string with an initial value of 'ABC'. &parm3 is declared an eight character string with an initial value of 'Hello'. The first Prompt statement executes, and the initial values of the three variables are displayed on the workstation screen. When the ENTER key is pressed, the Run statement executes procedure CALLEE, which is shown below. Once procedure CALLEE is completed, execution returns to the PROMPT statement following the RUN statement above. The values displayed by this PROMPT statement reflect the modifications made within procedure CALLEE.

Example

Procedure CALLEE, which receives parameters from CALLER

```
Using &l integer, &2 string(6), &3 string(1)
Assign &l = 100
Assign &2 = 'GEORGE'
Assign &3 = 'Q'
Prompt '&l = ', &l; '&2 = ', &2; '&3 = ', &3
```

Procedure CALLER passed actual parameters &parm1, &parm2, and &parm3 to procedure CALLEE, above. CALLEE received these parameters with the USING statement, and defined them as formal parameters &l, &2,

and &3. The Assign statements modify the formal parameters. The PROMPT statement displays the new values. When the ENTER key is pressed control returns to procedure CALLER, and the last PROMPT statement of procedure CALLER executes.

Procedures CALLER and CALLEE demonstrate the three types of parameter passing. The first parameter, &parm1, was passed by reference because it was a variable. The second parameter, &parm2, was passed by value-result because it was declared in CALLER as a three character string, but was received by CALLEE as a six character string. The third parameter, &parm3, was passed by value because it is a substring.

The following values were displayed by each PROMPT statement:

CALLER- 1st PROMPT

P1 = 5
P2 = ABC
P3 = H

CALLEE- 1st PROMPT

&1 = 100
&2 = GEORGE
&3 = Q

CALLER- 2nd PROMPT

P1 = 100
P2 = GEO
P3 = H

CHAPTER 8

ASSIGNING VALUES & DEFINING VARIABLES: DECLARE & ASSIGN

8.1 DECLARING VARIABLES: THE DECLARE STATEMENT

The DECLARE statement defines procedure variables. This statement also specifies the attributes of the variables and their initial values.

Example

```
PROCEDURE A
DECLARE &FIRST, &LAST AS STRING(6)
DECLARE &STATUS GLOBAL INTEGER INITIAL 100
RUN PROCB
```

Variables &FIRST and &LAST are declared for use in procedure A. Both are declared as type string with a length of 6. Since the INITIAL clause is not specified, the initial value of each variable is blanks. &STATUS is declared as global integer with an initial value of 100. &STATUS can be used in procedure A and in any procedures run directly or indirectly by A. If PROCB is a procedure, it can reference &STATUS without declaring it, but cannot reference &FIRST or &LAST.

8.2 ASSIGNING VALUES: THE ASSIGN STATEMENT

The ASSIGN statement is used to assign values to either a variable, or to a substring of a string-variable.

Example

```
procedure
declare &var1 integer initial 2
assign &var1 = &var1 + 1
```

In the example above, the value of &var1 is 2 before the ASSIGN statement is executed. The value of &var1 is 3 after the ASSIGN statement is executed.

Example

```
procedure
declare &data string(8) initial "ABCDEFGH"
assign &data(2,1) = "X"
assign &data(6,3) = "YYY"
assign &data(6,4) = "ZZZZ"
```

The example above demonstrates assignment to a substring of a string-variable. Before the first ASSIGN statement is executed, the value of &data is "ABCDEFGH". After the first ASSIGN statement is executed, the value of &data is "AXCDEFGH". After the second ASSIGN statement is executed, the value of &data is "AXCDEYYY". When the third ASSIGN statement is executed, an error occurs because the specified substring exceeds the bounds of &data.

CHAPTER 9

DISPLAYING AND ACCEPTING TEXT, VARIABLES AND CONSTANTS ON THE WORKSTATION

You can display or accept text, variables and constants on your workstation by using the PROMPT or MESSAGE statements.

MESSAGE and PROMPT statements are very similar. The basic difference between them is that PROMPT solicits a response from the user, and MESSAGE does not.

Section 9.1 discusses the MESSAGE statement in detail. All rules and syntax for MESSAGE also apply for PROMPT, unless otherwise noted. Section 9.2 provides information that is specific to the PROMPT statement.

9.1 DISPLAYING A MESSAGE ON THE WORKSTATION: THE MESSAGE STATEMENT

The MESSAGE statement displays text on the workstation and continues execution of the procedure. You cannot use this message to solicit information; the user cannot respond to this message. The message remains on the workstation until another MESSAGE or PROMPT statement is executed, a program or procedure that displays a message is run, or until the procedure terminates.

If MESSAGE is issued and the workstation is not available, or the procedure is running in the background, the MESSAGE statement is ignored.

9.1.1 Displaying Lines

You can create up to 24 lines of text for both MESSAGE and PROMPT. Each line can consist of up to 79 characters. Lines longer than 79 characters are automatically truncated from the right. If you create a message longer than 24 lines, an error occurs.

The semicolon separates each line of the message. If a line within a message is to be blank, a semicolon is used to denote the skipped line.

Example

Procedure

```
Message "This is a line of text.";
        "This is a second line of text."
```

When this procedure is run, it displays two lines of text which are centered vertically and left justified on the workstation. The message remains on the workstation only as long as the procedure is running.

If you specify "CENTER" for a line, the text on that line is horizontally centered. If you do not specify "CENTER" for a line, that line is left justified, beginning in column 2.

Lines are automatically centered vertically on the workstation. For example, if you create a message that is four lines long, those lines are displayed on lines 11 through 14 on the workstation. You can override this centering by using the ROW option to specify the first row where you wish the message to appear.

If the ROW option is specified, it must be followed by a value between 1 and 24 to indicate the first row where the message will begin. If the number of displayed lines, plus the value of the ROW option exceeds 25, then an error occurs.

Example

Procedure

Message ROW 2

"This is a line of text.";

CENTER "This is a second line of text."

When this procedure is run, it displays two lines of text. The first line begins on row 2 and is left justified. The second line begins on row 3, and is horizontally centered.

9.1.2 Creating Lines from Multiple Fields

You can use multiple fields to compose each line. A field is a string expression such as quoted text or variables. The fields of a line are separated by commas, and the lines are separated by semicolons.

Example

Procedure

Declare &name string(17)

Extract &name = username

Message center "My name is ", &name

When this procedure is run by user Christopher Robin, "My name is Christopher Robin" appears on the workstation. The message remains on the workstation only as long as the procedure is running.

9.1.3 Video Attributes

You can use various video attributes to indicate how fields are to appear on the screen. The following attributes are available:

<u>Attribute</u>	<u>Explanation</u>
BLINK	Field blinks with high intensity.
BRIGHT	Field appears with high intensity.
DIM	Field appears with normal intensity.
LINE	Field appears underlined, with normal intensity.
BLANK	Field is invisible.

Each attribute you specify applies only to the associated field. If you specify conflicting attributes (for example, BRIGHT and DIM in the same field), the last specified attribute applies. If you specify compatible attributes (for example, BRIGHT and LINE), all compatible attributes apply.

Example

Procedure

```

Declare &name string(17)
Extract &name = username
Message center "My name is ", &name;
               center LINE "Workstation I/O", "is", BLINK "easy", "to do!"

```

When this procedure is run by user Christopher Robin, the following message appears centered on the workstation screen:

```

My name is Christopher Robin
Workstation I/O is easy to do!

```

For the first line, no video attributes were specified for either field. Therefore, this sentence appears with normal intensity. The second sentence has been divided into four fields. The first field is underlined, and the third blinks with high intensity. The second and fourth fields have no attributes, and therefore appear with normal intensity.

When you specify attributes for at least one of two contiguous fields, one space is introduced between them. When you specify two contiguous fields, and neither has attributes, no space is introduced between them.

9.1.4 Using the ERASE Option to Build Messages

Each time a MESSAGE or PROMPT statement is executed, the entire workstation screen is erased prior to display of the message. Sometimes, you may wish to add text to an existing display. The ERASE option allows you to do this.

If ERASE = NO, the text of the message will be written directly to the workstation without the current contents being first erased. The new message is written to the workstation. Any existing workstation text that is not overwritten by the message will be left unaffected. If ERASE = YES or the ERASE option is not specified, the screen will be erased entirely just prior to the message display.

Example

Proc

```
Message           Row 17 "Message one";  
Message ERASE = NO Row 18 "Message two";  
Message ERASE = NO Row 17 "Message three"
```

When the first MESSAGE statement is executed, the workstation screen is erased, and "Message one" is displayed on row 17. When the second MESSAGE statement is executed, "Message two" is displayed on row 18 without affecting the first message. When the third MESSAGE statement is executed, "Message three" is displayed on row 17. "Message three" overwrites "Message one", but does not affect "Message two".

9.1.5 Ringling the Workstation Bell

You can use the ALARM option to ring the workstation bell when a message is displayed. This option can be used with either MESSAGE or PROMPT.

If ALARM = YES, The workstation bell rings when the message is displayed. If ALARM = NO, or is not specified, the bell does not ring.

9.2 SOLICITING A RESPONSE FROM THE WORKSTATION: THE PROMPT STATEMENT

The PROMPT statement is used to solicit a response from the user. You respond by supplying the requested information, and then pressing either ENTER or a specified PF key. After you press ENTER or a PF key, the "Procedure in progress" message is displayed, and execution of the procedure continues.

If PROMPT is issued and the workstation is not available, or the procedure is running in the background, an error is issued, and the procedure is canceled.

You create messages for the PROMPT statement in the same manner as the MESSAGE statement. Since the MESSAGE statement is a subset of the PROMPT statement, refer to Section 9.1 for information that describes both statements.

In addition to the capabilities of the MESSAGE statement, the PROMPT statement can display modifiable fields and receive the values the user enters into them. The PROMPT statement can also receive the PF key pressed, and the position of the cursor row and column.

9.2.1 Creating Modifiable Fields

Only integer variables, string variables, and substrings can be used for modifiable fields. A field is made modifiable by preceding it with one of three attributes: UPPER, UPLOW, or NUMERIC. The attribute used depends on the type of data you wish to receive.

You can specify modifiable fields by using the following attributes:

<u>Attribute</u>	<u>Explanation</u>
NUMERIC	Only numeric data accepted.
UPLOW	All characters accepted.
UPPER	Only uppercase characters accepted.

Modifiable fields appear on the workstation screen as pseudoblanks.

Example

Procedure

```
Declare &name string (24)
Declare &age integer
Prompt center "Enter your name and age, then press ENTER";;
           center "Name", UPLOW &name, "      Age" , NUMERIC &age
Prompt center "Your name is ", &name(1,*),
           " and you are ", &age, " years old"
```

When the first PROMPT statement is executed, the following message is displayed until the user presses ENTER or a PF key:

Enter your name and age, then press ENTER

Name ***** Age *****

The user can type his name into the modifiable field that follows the word "Name." Since the UPLOW attribute was specified, any characters can be typed into the field. If George Washington types in his name and age, and presses ENTER, the second PROMPT statement displays the following message:

Your name is George Washington and you are 252 years old

This message contains no modifiable fields. It remains on the screen until the user presses ENTER or a PF key.

When a string variable is used for a modifiable field, the displayed length of the field is the length of the variable. When a substring is used for a modifiable field, the displayed length of the field is the length of the substring.

When an integer variable is used for a modifiable field, the displayed length of the field is eleven. Eleven positions allows room for the full range of integer values, -2147483647 to 2147483647. Values entered outside of this range cause the field to blink and the workstation bell to ring. You can enter integer data with leading or trailing signs but there can be no space between the sign and the number. You cannot use UPPER or UPLOW with an integer variable.

If you display the same modifiable field more than once in a single prompt, the value you supply at the workstation to the first field (the uppermost, leftmost, duplicate, and modifiable field on the screen) is used for the assignment.

9.2.2 Receiving PF Key, Cursor Row and Cursor Column

After a user responds to a PROMPT screen, you can determine which PF key the user pressed. You can also determine the cursor location at the time a PF key was pressed.

If you specify the PFKEY clause, the associated variable will receive the PF key number (1 to 32, or 0, if ENTER is pressed) at the time ENTER or a PF key is pressed.

If you specify the CURROW clause, the associated variable will receive the row position of the cursor (1 to 24) at the time ENTER or a PF key is pressed.

If you specify the CURCOL clause, the associated variable will receive the column position of the cursor (1 to 80) at the time ENTER or a PF key is pressed.

Example

Procedure

```
Declare &pfkey, &row, &column integer
Prompt PFKEY = &pfkey, CURROW = &row, CURCOL = &column
      center "Position the cursor as desired,";
      center "then press ENTER or any PF key"
Prompt center "The cursor was at row ", &row, " column ", &column,
      " when you pressed pfkey ", &pfkey
```

When the user responds to the first PROMPT, the value of the PF key pressed is stored in &pfkey, and the values of the cursor row and column are stored in &row and &column, respectively. The second PROMPT displays these values.

It is not possible to disable PFKEYS. Therefore, when writing procedures using the PFKEY option, you should make sure that the procedure's logic can handle or ignore undesirable PF keys.

9.2.3 Tabs

You can create fields that can be tabbed to, but are not modifiable. This feature is useful when creating certain types of menus.

When you specify TAB as the attribute of a field, one pseudoblink appears in front of the field. You can tab to the pseudoblink, but you cannot modify it.

Example

Procedure

```
      Declare &cursor_row integer
Loop:  Prompt  currow = &cursor_row  row 10
          TAB " Run COPY";
          TAB " Run DISPLAY";
          TAB " Exit"
      If &cursor_row = 10 then run COPY
      If &cursor_row = 11 then run DISPLAY
      If &cursor_row = 12 then return
      Goto loop
```

This procedure results in the following output:

```
* Run COPY
* Run DISPLAY
* Exit
```

The user can respond to this PROMPT by using the TAB key to position the cursor to the desired function. When the user presses ENTER, or any PF key the specified function is performed. If the cursor is not on row 10, 11, or 12, no function is performed.

If you specify TAB, you cannot specify any other attribute for that field, such as BRIGHT or LINE. However, you can specify other options, such as CENTER, ROW or ERASE.

CHAPTER 10

THE SYNTAX CHECKER

The Syntax Checker flags most of your syntax errors in a single run, without executing the procedure. When the Syntax Checker detects an error, a message is written to an error file.

Syntax checking is one of the Procedure Interpreter modes. In this mode, the interpreter functions in the following manner:

1. No statements are executed.
2. Errors and warnings are not displayed at the workstation until you display the error file where they are stored.
3. When an error is encountered, the interpreter skips the rest of the statement containing the error. It resumes interpretation with the next label definition or verb in the source text.

The Syntax Checker makes a distinction between high and low severity errors. If a severity level is four or below, this error is a warning, or a low severity error. If a statement contains only warnings, the syntax check continues. Any error with a severity level above four is a fatal error, and the rest of that statement is skipped. The syntax check resumes with the next label or verb.

Example

```
procedure
run editor
display input LANGUAGEE = procedure, FILE = demo
```

The example procedure above contains a typing error. If you were to run the Syntax Checker with this procedure, a warning message would be written to the error file. The message would indicate that keyword LANGUAGEE contains more than 8 characters and that only the first 8 characters will be used. Since this message is only a warning, the syntax check would continue to check this statement. If the warning were an error, the rest of the statement would be skipped, and the syntax check would resume with the next statement.

Example

```
procedure
declare &var1 strxng (9)
assign &var1= "ROCKY"
```

If you were to run the Syntax Checker with the procedure above, the following errors would be written to the error file:

```
declare &var1 strxng (9)
ERROR 13 (column 15):
Expecting required keyword INTEGER or STRING, but found STRXNG.
```

```
assign &var1= "ROCKY"
WARNING 76 (column 8):
Variable &VAR1 has not been declared or is declared globally outside
of this procedure. It was declared by the Procedure Interpreter as a
string(256) variable so that syntax checking could continue.
```

Since keyword STRING was misspelled, the Procedure Interpreter issued an error for the DECLARE statement. The rest of the DECLARE statement was skipped, and the syntax check continued with the next statement. Since the interpreter did not complete its check of the DECLARE statement, variable &VAR1 was not recognized when it was encountered in the ASSIGN statement and a warning was issued.

Since one error can cause multiple errors, you might find it helpful to run the Syntax Checker after each syntax modification, until no error messages appear.

If you do not run your procedure through the Syntax Checker before execution, all errors flagged by the Procedure Interpreter are considered severe. A procedure that contains no high severity errors will run even if it contains warnings.

10.1 RUNNING THE SYNTAX CHECKER

You can run the Syntax Checker in two ways: from the EDITOR, or from the PROC program. Each of these methods is discussed in the following subsections.

10.1.1 Syntax Checking from the EDITOR

To run the Syntax Checker from the editor, select (10) from the special menu (Syntax Check). The screen displayed in figure 10-1 appears.

```
*** Wang VS Integrated Program Development EDITOR - ***
There is no current input file.
There are 23 lines in the edited text. (The file has been modified.)

Please select the desired function and press the appropriate PF key:

(1) Display      - Resume text editing
(2) Set          - Set workstation defaults and Editor/tracing options
(3) Menu         - Activate the normal menu
(4) Restart      - Edit another file
(5) Create       - Create a new file from the edited text

(7) Renumber     - Generate new line numbers for the edited text
(8) External     - Copy a range of lines from another file to the work
(9) Run          - Run the procedure
(10) Syntax Check - Check the edited text for correct syntax

(13) Utilities   - Run VS utility (or other) programs

(15) Print       - Print the edited text
(16) Exit        - End EDITOR processing
```

Figure 10-1 The Special Menu Screen from the EDITOR

When you press PF key 10, a message appears indicating that a syntax check is in progress. When the check is complete, the EDITOR main menu returns. If no syntax errors were detected, the message "Syntax check of ##TEST completed" appears. Otherwise, the message is "Syntax check of ##TEST completed with errors- severity level is ____." The severity level appears in the blank.

If the syntax check was completed with errors, the special menu screen includes an option to display the syntax errors. Press PF key 11 to read the error display.

Figure 10-2 shows a sample error file. Each error lists the statement line number and the statement containing the error. The line below the statement contains the error number, the location of the error, and a message to explain the error.

```
(1)Disp (2)First (3)Last (4)Prev (5)Next (6)Down (7)Up (8)Find Error  
(16)Menu
```

```
000200 declare &var1 strxng (9)  
      ERROR 13 (column 15):  
      Expecting required keyword INTEGER or STRING, but found STRXNG.  
000300 assign &var1= "ROCKY"  
      WARNING 76 (column 8):  
      Variable &VAR1 has not been declared or is declared globally outside  
      of this procedure. It was declared by the Procedure Interpreter as a  
      string(256) variable so that syntax checking could continue.
```

Figure 10-2 Sample Error File

After you have read the error messages, press PF key 1 to display your procedure. You can now modify your procedure by pressing PF key 9 and then altering your input.

10.1.2 Running the Syntax Checker From the PROC Program

You can run the Syntax Checker as a separate program. To check the syntax of a procedure, run PROC in the system library on the system volume. Figure 10-3 shows the first screen of the PROC program.

```
Wang VS Procedure Interpreter -- Version 03.00.03          Screen: OPTIONS
-----
Enter procedure name and select:
File ***** in library ***** on volume *****

(1) Run    - Run procedure or program
(2) Edit   - Edit file

(9) Trace  - Run procedure and trace execution
(10) Check - Check procedure for correct syntax

(14) Display - Display file
(16) Exit   - End processing
```

Figure 10-3. The PROC Options Screen

To run the Syntax Checker from PROC, specify the file name and location, then select CHECK by pressing PF key 10.

Two messages are displayed on the workstation, one after the other. The first message appears, and indicates that a syntax check is in progress. The second message replaces the first, and appears at the bottom of the PROC Options screen when the syntax check is completed. If no syntax errors were detected, the message "Syntax check of ##TEST completed" appears. Otherwise, the message is "Syntax check of ##TEST completed with level _ errors." The severity level appears in the blank.

To view the error file, select Errors by pressing PF key 11 from the PROC option screen.

You can use a procedure to run the PROC program. The prname for the main screen is OPTIONS.

CHAPTER 11

THE TRACING FACILITY

The Tracing Facility creates a record of Procedure Interpreter activity. The tracing facility provides you with the ability to

- Trace all statements, or trace RUN statements only.
- Specify the class of information to be traced.
- Perform high-level debugging of single and nested procedures. Tracing depicts the procedure flow of control across link levels.
- Perform low-level debugging of procedures by recording run-time value of variables used within statements.

11.1 HOW THE TRACING FACILITY FUNCTIONS

The tracing facility starts a trace at the link level of the procedure that initiates it. The trace continues at that level, and at all subsequent link levels until tracing is explicitly terminated, or until the procedure that initiated it has been completed. Tracing can only terminate at the same link level at which it was initiated; it cannot be terminated at a subsequent link level.

Tracing also occurs through execution of object programs. For example, if Procedure A initiates tracing and then runs object program B, which in turn runs Procedure C, then tracing of statements in Procedure C also occurs.

Only one trace file at a time can be active for a single task. As a result, tracing commands cannot be nested.

11.2 RUNNING THE TRACING FACILITY

There are three methods of running the trace facility:

1. You can code a TRACE statement into your procedure.
2. You can initiate it from the EDITOR's Special Menu screen.
3. You can run it from the PROC program.

11.2.1 Tracing from a Procedure

You can use the TRACE statement to initiate or terminate tracing. The general format for the TRACE statement is described in chapter 12.

You initiate tracing by coding the TRACE verb, and optionally coding the INTO, RESOURCES, SCRATCH, STATEMENTS, and VARIABLES clauses. You terminate tracing by coding TRACE END.

You use the INTO clause to specify the name of the trace file. All trace information is written to this file. If you are initiating tracing, and you do not use the INTO clause, the default trace file name is the same as the current procedure, and the library and volume are your SPOOLIB and SPOOLVOL. If you specify the INTO clause, and omit the library and/or volume, the default library and volume are your SPOOLIB and/or SPOOLVOL.

The TRACE statement has four options. You can specify these options when you initiate tracing. You can also change these options while tracing is in progress. To change an option while tracing is in progress, you code the TRACE statement without the INTO clause, and specify the options you want to change.

TRACE options remain as specified or defaulted until you explicitly change them, you terminate the trace by coding TRACE END, or the procedure that initiated tracing terminates.

Example

Proc

```
TRACE INTO test1 in gstrace on system, STATEMENTS = RUN
```

```
.
```

```
.
```

```
TRACE, STATEMENTS = ALL
```

When the first TRACE statement is executed, tracing is initiated and the trace information is written to file TEST1 in GSTRACE on SYSTEM. Since STATEMENTS = RUN was specified, only RUN statements located between the first and second TRACE statement are traced. After the second TRACE statement is executed, all statements will be traced, since the STATEMENTS option was changed to ALL. The TRACE information continues to be written to file TEST1.

You can respecify the TRACE file without terminating and reinitiating the trace. You do this by including an INTO clause in a second TRACE statement. If in the previous example, the second TRACE statement read

```
TRACE INTO test2 in gstrace on system, STATEMENTS = ALL
```

then tracing would have been altered to include all statements, but the trace information would be written to file TEST2. If tracing is in progress, and the INTO clause is used, the current trace file is closed, and the new trace file is opened. Therefore, file TEST1 is closed and then TEST2 is opened.

You can use the SCRATCH option to scratch the specified trace file before it is opened. This option is especially useful if you trace a procedure more than once, and wish only to keep the latest trace. If you do not specify the SCRATCH option, or you specify NO, the trace listing is appended to the trace file, if one exists. A trace file is scratched only when the INTO clause is specified.

You can use the RESOURCES option to trace resource usage statistics. These statistics are the CPU time and elapsed time for programs or procedures run with the RUN statement. If you specify RESOURCES = YES, resources will be traced; otherwise, no trace is made.

You can use the STATEMENTS option to specify tracing of RUN statements only. If you specify STATEMENTS = RUN, only RUN statements are traced; otherwise, all statements are traced.

You can use the VARIABLES option to trace the post execution run-time values of variables. These values reflect the contents of a variable after the statement containing that variable has been executed. If you specify VARIABLES = YES, the values are traced; otherwise, the values are not traced.

Example

Proc

```
Trace, VARIABLES = YES
Declare &X integer initial 5
Assign &X = &X + 1
```

When the ASSIGN statement of this procedure is traced, the trace listing will show the value of &X as 6, because this is the value of &X after the statement is executed. If you did not specify the VARIABLES clause, this information would not be recorded.

11.2.2 Running the Tracing Facility from the EDITOR

You can perform tracing from the Special Menu of the EDITOR. From the special menu, select the Set Workstation Defaults and Editor/Tracing Functions by pressing PF key 2. The Set command screen shown in Figure 11-1 appears.

Set Command

Please select the desired function and press the appropriate key.
Press PF1 to return to the special command menu.

EDITOR Options

- (2) Set tabs, upper/lower case and search modes
- (3) Set automatic scratch, renumber, and replace modes and modification code

Syntax Checking and Tracing Options

- (4) Change Tracing Options
- (5) Change default file, library and volume names for error listing and tracing files.

Figure 11-1 Set Command Screen

From the SET screen, select the Tracing Option by pressing PF key 4. The Tracing Definition Screen, shown in Figure 11-2, appears.

```
*** MESSAGE 0020 BY EDITOR

      INFORMATION REQUIRED BY PROCEDURE EDIT
      TO DEFINE TRACING

Enter tracing options as desired. Press PF1 to exit without
setting options.

Trace the execution of this procedure?      TRACE   = NO* (YES/NO)

If TRACE=YES,
Trace only RUN statements?                  RUNONLY = NO* (YES/NO)
Trace values of variables?                 VARIABLE= NO* (YES/NO)
Trace resource usage?                      RESOURCE= NO* (YES/NO)

Trace within line range:                   START   = ALL***
                                           END      = *****
```

Figure 11-2 Tracing Definition Screen

Enter "RUNONLY= YES" if you wish to trace only the RUN statements. Otherwise, the default is NO, and all statements will be traced.

Enter "VARIABLES= YES" if you wish to trace the post statement execution run-time values of variables. The default for VARIABLES is NO.

Enter "RESOURCE= YES" if you wish to trace the resource usage statistics for the RUN statements. The default for RESOURCE is NO.

If you wish to trace only certain lines, fill in the range of lines desired. START specifies the first line of the trace, and END specifies the last line. If you wish to trace the whole procedure, let the default "ALL" remain.

Press ENTER once your specifications are complete. Press PF key 1 to return to the main menu.

You are now ready to run your procedure. During the run, a trace file will be created or the new listing will be appended. Once the run is complete, you can select PF key 12 from the EDITOR's special menu to display the trace listing.

11.2.3 Running the Tracing Facility as Part of the PROC program

You can run the Tracing Facility from the PROC program. To trace a procedure, run PROC in the system library and volume. Figure 11-3 shows the Options screen of the PROC program.

```
Wang VS Procedure Interpreter -- Version 03.00.03          Screen: OPTIONS
-----
Enter procedure name and select:
      File ***** in library ***** on volume *****

      (1) Run    - Run procedure or program
      (2) Edit   - Edit file

      (9) Trace  - Run procedure and trace execution
      (10) Check - Check procedure for correct syntax

      (14) Display - Display file
      (16) Exit   - End processing
```

Figure 11-3. The PROC Options Screen

To run the tracing facility from PROC, fill in the filename and location, then select the Trace option by pressing PF key 9. The TRACE screen, shown in Figure 11-4, appears.

Specify trace options and press (ENTER):

Trace Run statements only?	Runonly	= NO*	(YES/NO)
Trace values of variables?	Variable	= NO*	(YES/NO)
Trace resource usage?	Resource	= NO*	(YES/NO)
Trace within line range:	Start	= ALL***	
	END	= *****	

(ENTER) Accept - Accept options and begin trace
(1) Return - Return to OPTIONS screen
(16) Exit - End processing

Figure 11-4. The TRACE Screen

Enter "RUNONLY= YES" if you wish to trace only the RUN statements. Otherwise, the default is "NO", and all statements will be traced.

Enter "VARIABLES= YES" if you wish to trace the post statement execution run-time values of variables. The default for VARIABLES is "NO" and the post statement execution trace will not occur.

Enter "RESOURCE= YES" if you wish to trace the resource usage statistics for the RUN statements. The default for RESOURCE is "NO", and the statistics will not be traced.

If you wish to trace only certain lines, fill in the range of lines desired. START specifies the first line of the trace, and END specifies the last line. If you wish to trace the whole procedure, let the default "ALL" remain.

Press ENTER once your specifications are complete.

Once the tracing is completed, the PROC Options screen reappears, with an option to display the trace listing. Press PF key 12 to read this listing.

11.3 A SAMPLE PROCEDURE AND TRACE LISTING

The following procedures demonstrate the TRACE statement:

```
000100 Procedure PROC1
000200 Trace into demo, variables = yes
000210 declare &file, &library string(8)
000310 run PROC2 using &file, &library
000320 trace, resources = yes
000410 run copy
000510 display input file = &file, library = &library, volume = system
```

```
000100 Procedure PROC2
000200 Using &parml, &parm2 string(8)
000300 Assign &parml = "DEMO"
000310 Assign &parm2 = "testlib"
000400 Return code = 80
```

When PROC1 is run, the trace listing in Figure 11-5 will be created and stored in file DEMO in the user's SPOOLIB, on the user's SPOOLVOL.

The trace listing is divided into four columnar sections, as follows:

<u>Columns</u>	<u>Information contained</u>
1 - 2	Procedure nesting level
5 - 12	Time stamp
15 - 80	Statement line number and text
82 - 131	Statement data

The following subsections discuss each section.

11.3.1 Procedure Nesting Level

The procedure nesting level distinguishes between procedures that are running at different levels within the scope of the trace. If you code a TRACE statement, the procedure containing the TRACE statement is at level one. If you are tracing from the EDITOR or from PROC, the procedure you run is at level one. A procedure run from the level one procedure is at level two.

The level number indicates the number of procedures that are nested. No level number is assigned to programs. For example, if procedure A initiates tracing and then runs program B, which in turn runs procedure C, then procedure A will be at level one, and procedure C will be at level two, not at level three.

```

1      14:40:00 000210 DECLARE &FILE , &LIBRARY STRING ( 8 )
1      1
1      1
1      14:40:00 000310 RUN PROC2 USING &FILE , &LIBRARY
1      1
2      14:40:01 000200 USING &PARM1 , &PARM2 STRING ( 8 )
2      1
2      14:40:01 000300 ASSIGN &PARM1 = "DEMO"
2      14:40:01 000310 ASSIGN &PARM2 = "testlib"
2      14:40:01 000400 RETURN CODE = 80
1      1
1      1
1      14:40:02 000320 TRACE , RESOURCES = YES
1      14:40:02 000410 RUN COPY
1      DISPLAY INPUT FILE = &FILE , LIBRARY = &LIBRARY, VOLUME =
1      SYSTEM
1      1
1      1
1      Return code = 0
1      CPU time = 00:00:00:10 Elapsed time = 00:00:02
1      &FILE = "DEMO"
1      &LIBRARY = "testlib"
1      *Tracing terminated.

```

In the listing shown in this section, procedure PROC1 is at level one because it initiated tracing. Procedure PROC2 is at level two because it was run from PROC1.

11.3.2 Time Stamp

The time stamp shows the time that the execution of each statement began. The time is shown in the format HH:MM:SS, where HH is the hour, MM is the minute, and SS is the second.

11.3.3 Statement Line Number and Text

Columns 15 through 20 of this section contains the source line number of each statement being traced. This is the same line number found in the procedure source file.

Columns 22 through 80 contain the text of each statement being executed. The text appears as it was viewed by the Procedure Interpreter, not as it appears in the actual source file. One space appears between each element of the text. All unquoted text is shown in uppercase. Quoted text is shown as it was coded. Text is continued on successive lines, as necessary.

The order of statements shown in the trace listing reflect the order of statements as they were executed. In the trace listing shown in this section, the first statement shown is the DECLARE statement of PROC1 (the TRACE statement, which initiated tracing, was not traced). The second statement is the RUN statement of PROC1. The third statement shown in the trace listing is the USING statement of PROC2 because it is the next statement executed. The procedure nesting level numbers on the far left side of the listing help to distinguish between the two procedures.

11.3.4 Statement Data

This section of the listing contains messages about the trace, and data about the statements executed.

Messages are preceded by asterisks, and provide general information about the trace. In the sample listing provided, the first and last messages indicate that tracing initiated and terminated. The second and third messages help distinguish between PROC1 and PROC2.

The data section also shows statement return codes, variable values, and CPU and elapsed time. Return codes are always shown for statements that provide return codes. Variable values are only shown if VARIABLES = YES was specified. CPU and elapsed time are shown only if RESOURCES = YES.

Variable values appear as the variable name followed by an equal sign, followed by the value enclosed in quotes. If necessary, the value is continued on the following line. When this occurs, a hyphen appears in column 81, to indicate continuation. If the value contains unprintable characters, the characters appear as blanks.

CPU time is shown in the format HH:MM:SS:th, where HH is the hour, MM is the minute, SS is the second, t is tenths of a second, and h is hundredths of a second.

Elapsed time is shown in the format HH:MM:SS, where HH is the hour, MM is the minute, and SS is the second.

PART II

Statement and Built-In Function Reference

CHAPTER 12

PROCEDURE LANGUAGE STATEMENT REFERENCE

12.1 CONVENTIONS USED IN THIS REFERENCE

Each statement described in this chapter includes the following:

Function The Function section describes the effect(s) of statement use.

General Format The General Format section presents the statement and its elements as they appear in a program. The elements that make up the general format are uppercase words, lowercase words, and syntactic symbols.

Uppercase words are keywords.

Example: USING variable

USING is a keyword.

Lowercase words are generic terms that you must replace with appropriate names or values.

Example: CODE = integer-expression

integer-expression must be replaced with an integer expression when the CODE option is used.

Underlined words are the default values.

Example: OPTIONS PROCMSG = $\left\{ \begin{array}{l} \underline{\text{YES}} \\ \text{NO} \end{array} \right\}$

If PROCMSG is not specified, the default value is YES.

Syntactic symbols are used to enclose portions of the general format. Each symbol is explained below.

[] Optional - contents are optional. The information contained may be omitted. If you include the contents, you can use only one of the options.

Example: MOUNT [DISK
TAPE]

Zero or one, but not both, of the options
- DISK, TAPE can be specified.

{ } Pick one - surrounding values indicates that you must select one value. No more than one value can be selected.

Example: ,CPULIMIT = hh:mm:ss
ss

Either hh:mm:ss, or ss must be specified, but not both.

... Repeat - three consecutive periods indicate that you can repeat the portion of the statement surrounded by the immediately preceding paired symbols.

Example: [, variable]...

Any number of variables, separated by commas, can be specified.

Syntax Rules

This section presents the syntax rules required for correct formation of the statement.

General Rules

The General Rules section describes the rules and restrictions governing statement use. Operand descriptions appear in the same order as the components in the general format.

Example

Each statement description includes an example that demonstrates correct syntax and use of the statement.

Although keywords or keyword parameters appear in uppercase in this manual, you can enter them in your procedure as lowercase. You can also enter the user-supplied identifier names in lowercase or uppercase.

12.2 ASSIGN

Function

The ASSIGN statement assigns values to a variable or to a substring of a string-variable.

General Format

$$[\text{label:}] \dots \text{ASSIGN } \left\{ \begin{array}{l} \text{variable} \\ \text{substring} \end{array} \right\} = \text{expression}$$

General Rules

1. The ASSIGN statement assigns the value of the expression to the named variable or substring. There are six possible assignments:
 - a) Integer expression to integer variable. The expression is evaluated and the resulting value is stored in the integer variable.
 - b) String expression to integer variable. The string must not exceed 16 characters, with no more than 11 characters as signs or digits. This string can begin with optional leading blanks, followed by an optional sign, followed by the digits of the integer, followed by optional trailing blanks. If the size or format rules are violated, or if the string does not represent an integer, an error is issued. The string is converted to its integer value and is stored in the integer variable.
 - c) Boolean expression to integer variable. Expression is evaluated to either 0 or 1. The result is stored in the integer variable.
 - d) Integer expression to string variable or substring. The expression is evaluated and converted to a string and stored, left-justified, in the string variable or substring. The string consists only of the significant digits in the integer, and if negative, is preceded by a minus sign. If the integer is zero, the string receives a single '0'. If the receiving field is shorter than the sending field, truncation occurs from the right. If the receiving field is longer than the sending field, the receiving field is padded with blanks.

- e) String expression to string variable or substring. The expression is stored, left-justified, in the string variable or substring. If the receiving field is shorter than the sending field, truncation occurs from the right. If the receiving field is longer than the sending field, the receiving field is padded with blanks.
 - f) Boolean expression to string variable or substring. The expression is converted to either 0 or 1, and is stored in the string variable left-justified. If the receiving field is longer than the sending field, the receiving field is padded with blanks.
2. A substring on the left of the equals (=) sign refers to the character positions in the variable to which the assignment is made. A substring on the right of the = refers to the contents of those character positions.
 3. Unquoted strings in expressions are interpreted as label references.
 4. label: is a string of one to eight alphanumeric characters that must begin with an alphabetic character and be immediately followed, with no intervening spaces, by a colon. The label can be the target of a GOTO statement. The ASSIGN statement does not assign a return code to the label.

Example

Procedure to show different types of assignments

```
label1: declare &long_string as string(15)
label2: declare &short_string as string(3)
label3: declare &integer as integer
```

*Integer expression to integer variable

```
assign &integer = 1 + 2           [assigns 3]
assign &integer = label2         [assigns value of label2]
```

*String expression to integer variable

```
assign &integer = 2 !! "3"       [assigns 23]
assign &integer = "0017"         [assigns 17]
assign &integer = " -123"        [assigns -123]
assign &integer = " - 123"       [error: space after sign illegal]
assign &integer = "00000002147483647" [error: string > 16 characters]
assign &integer = "123h5"        [error: can't convert string]
assign &integer = "123.5"        [error: decimal point illegal]
```

*** Boolean expression to integer variable**

```
assign &integer = 1 > 2          [assigns 0 (false)]
assign &integer = "abc" != "d" = "abcd" [assigns 1 (true)]
```

***Integer expression to string variable**

```
assign &short_string = 12345      [assigns "123"]
assign &long_string = 12345       [assigns "12345"    "]
assign &long_string = -12345      [assigns "-12345"   "]
assign &long_string = +12345      [assigns "12345"    "]
assign &long_string = 000000      [assigns "0"       "]
```

***String expression to string variable**

```
assign &short_string = "Text"     [assigns "Tex"]
assign &long_string = "Text"      [assigns "Text"  "]
```

***Boolean expression to string variable**

```
assign &short_string = 1 eq 2      [assigns "0  "]
assign &long_string = 1 lt 2       [assigns "1"   "]
```

***Assignments to substringed variables**

```
declare &var1, &var2, &var3 string(10)
assign &var1(3) = "abcde"          [&var1 = "  abcde  "]
assign &var2(5,2) = "wxyz"         [&var2 = "    wx   "]
assign &var3 = "$$$$$$"           [&var3 = "$$$$$$  "]
assign &var3(3,*) = '#####'       [&var3 = "$$####  "]
```

12.3 DECLARE

Function

The **DECLARE** statement defines the variables used within the procedure. It defines the type, size, initial value, and scope of variables.

General Format

```
[label:]...DECLARE variable [,variable] ...[AS] [GLOBAL]
      { STRING (expression) }
      { INTEGER               } [INITIAL expression]
```

General Rules

1. The **DECLARE** statement defines the listed variables. The presence or absence of **GLOBAL** defines the scope of reference for the variables. **STRING** and **INTEGER** define the type of the variables. **INITIAL** specifies the initial value of the variable.
2. When multiple variables are declared in the same statement, they all have the same type, length, and initial value.
3. If **GLOBAL** is specified, the variable can be referenced in the procedure in which it is declared, and all procedures called directly or indirectly by the procedure. Once the procedure in which the variable is declared terminates, the variable can no longer be used. The variable is called a global variable.

If **GLOBAL** is not specified, the variable can be referenced only in the procedure in which it is declared. The variable is called a local variable.

If a global variable is declared, and another global variable with the same name but different type or size already exists, an error occurs.

If a global variable is declared, and another global variable with the same name, type and size already exists, the declaration for the duplicate variable is ignored. However, any declarations within the same statement for non-duplicate variables will process normally.

If a local variable is declared, and a global variable having the same name already exists, the variable is declared a local variable and the global variable with the same name is no longer accessible.

When a variable is referenced, the Procedure Interpreter first looks for a local declaration of that variable, and uses it if found. If no local declaration exists, the Procedure Interpreter looks for a global variable, and uses it if one exists. If neither a local nor a global variable exists, an error occurs.

4. **STRING** is used to define a string variable. The expression is an integer expression whose value is between 0 and 256. This value defines the size of a string variable.

INTEGER is used to define an integer variable.

5. If **INITIAL** is specified, the value of the expression is assigned to the variable. The value must agree in type with the variable, or conversion to the variable's type must be possible.

If **INITIAL** is not specified, the default initial value for an integer variable is zero, and the default initial value for a string variable is all blank.

6. A variable cannot be referenced until the **DECLARE** statement containing it has been executed.
7. Variables can be redeclared. If duplicate declarations are made, the Procedure Interpreter uses the most recently executed declaration.
8. Strings used in the expression of the **INITIAL** clause must be quoted. Label references cannot be used.
9. **label:** is a string of one to eight alphanumeric characters that must begin with an alphabetic character and be immediately followed, with no intervening spaces, by a colon. The label can be the target of a **GOTO** statement. The **DECLARE** statement assigns a return code to the label. Refer to Appendix A of this manual for the return code values.

Example

Procedure to demonstrate the Declare statements

***Invalid examples**

<code>declare & as integer</code>	[< 2 characters]
<code>declare &this_is_too_long_for_a_var_name integer</code>	[> 31 characters]
<code>declare var string(5)</code>	[missing '&']
<code>declare &var_?_name integer</code>	[invalid character]
<code>declare &y as string(257)</code>	[length too large]
<code>declare &volume string(6) initial VOL500</code>	[bad initial value]
<code>declare &rtncode integer initial label</code>	[bad initial value]

***Valid examples**

```
declare &volume string(6) initial "VOL500"  
declare &i, &j, &k integer  
declare &null as string(0)  
declare &long as string(256)  
declare &counter global integer initial 1  
declare &file, &library global string(8)
```


12.4 DESTROY

Function

The **DESTROY** statement closes the currently executing procedure file, attempts to scratch it, and returns control to the invoking routine.

General Format

[label:]... **DESTROY PROCEDURE**

General Rules

1. The currently executing procedure file is closed, an attempt is made to scratch the procedure file, and control is returned to the invoking routine (which can be a program, a procedure, or the Command Processor). The return code passed back to the invoking routine is that resulting from the scratch. Refer to Appendix A of this manual for the return code values from scratch.
2. label: is a string of one to eight alphanumeric characters that must begin with an alphabetic character and be immediately followed, with no intervening spaces, by a colon. The label may be the target of a GOTO statement. The **DESTROY** statement does not assign a return code to the label.

Example

Procedure **TEMP** to be destroyed after execution

```
RUN TEST IN WORKLIB ON FLOPPY  
DESTROY PROCEDURE
```

12.5 DISMOUNT

Function

The DISMOUNT statement logically dismounts a disk or a tape.

General Format

Format 1

```
[label:]... DISMOUNT [DISK] {volname}
                             {(label)}
```

Format 2

```
[label:]... DISMOUNT TAPE {volname}
                           {(label)}
```

General Rules

1. Format 1 is used to dismount the disk volume specified by volname.
2. Format 2 is used to dismount the tape volume specified by volname.
3. volname is a one to six character string consisting of the letters A through Z, @, #, \$, or 0 through 9 that identifies a VS volume. This can be specified as a constant, a variable, a partial backward reference, or any other string expression. If the length of the string exceeds six, only the first six characters are used.
4. (label) can be used as a backward reference instead of specifying the volume name. The label must reference a DISPLAY or ENTER clause of a RUN statement that has been executed.
5. Unlike the DISMOUNT command issued interactively through the Command Processor, the DISMOUNT statement does not generate an error message if the specified volume cannot be dismounted.
6. Unquoted strings are interpreted as upshifted strings. For example, Label is interpreted as "LABEL".
7. label: is a string of one to eight alphanumeric characters that must begin with an alphabetic character and be immediately followed, with no intervening spaces, by a colon. The label can be the target of a GOTO statement. The DISMOUNT statement assigns a return code to the label. Refer to Appendix A of this manual for the return code values.

Example

Procedure to demonstrate MOUNT and DISMOUNT
MOUNT DISK FLOPPY ON 57 WITH NO LABEL
RUN DISKINIT
ENTER FUNCTION FUNCTION=INITIALIZE, VOLUME=FLOPPY
ENTER INPUT LABEL=NL
ENTER EOJ 16
DISMOUNT DISK FLOPPY

When this procedure is run, you are requested to mount a nonlabeled disk named FLOPPY onto device numbered 57. Next, the DISKINIT program runs, and disk FLOPPY is initialized. After DISKINIT is completed, you are requested to dismount the disk.

12.6 EXTRACT

Function

The **EXTRACT** statement extracts information from the system and stores it in the variables specified in the statement.

General Format

Format 1

```
[label:]... EXTRACT variable = keyword [, variable = keyword] ...
```

Format 2

```
[label:]... EXTRACT variable = { BLOCKS ALLOCATED FOR  
                                { RECORDS USED BY  
                                { filename [IN libname] [ON volname]  
                                { (label)
```

Syntax Rules

1. For Format 1, the keywords identifying fields from which you can extract data are as follows:

CURLIB	LINES	PRTFCLAS	TASK#
CURVOL	OTIO	RUNLIB	TASKTYPE
DISKIO	OUTLIB	RUNVOL	USERID
FILECLAS	OUTVOL	SPOOLIB	USERNAME
FORM#	PRINTER	SPOOLSYS	VERSION
INLIB	PRINTIO	SPOOLVOL	WORKLIB
INVOL	PROGLIB	SYSLIB	WORKVOL
JOBCLASS	PROGVOL	SYSVOL	WS
JOBLIMIT	PRNTMODE	SYSWORK	WSIO
JOBQUEUE	PRTCLASS	TAPEIO	

Refer to Section 5.2 for the definitions of these keywords.

2. A string variable can be specified to receive data for all of the keywords listed in Syntax Rule 1. An integer variable can be specified to receive data for the following keywords:

DISKIO	OTIO	PRINTIO	WS
FORM#	PRINTER	TAPEIO	WSIO
LINES	PRINTER	TASK#	

General Rules

FORMAT 1

1. The information for the specified keywords is extracted from the system and stored in the specified corresponding variable.
2. If PRINTER is extracted and returns a value of negative one (-1), the printer number specified on the Set Usage Constants screen of the command processor (the task's default printer) does not exist.

FORMAT 2

3. If BLOCKS ALLOCATED is specified, the Procedure Interpreter assigns to the variable the number of blocks allocated for the referenced file. If the file does not exist, the value of -1 is assigned.
4. If RECORDS USED is specified, the Procedure Interpreter assigns to the variable the number of records used by the referenced file. If the file does not exist, a value of negative one (-1) is assigned.
5. filename is a one to eight character string consisting of the letters A through Z, @, #, \$, or 0 through 9 that identifies a VS filename. This can be specified by a constant, a variable, a partial backward reference, or any other string expression. If the length of the string exceeds eight, only the first eight characters are used.

libname is a one to eight character string consisting of the letters A through Z, @, #, \$, or 0 through 9 that identifies a VS library. This can be specified by a constant, a variable, a partial backward reference, or any other string expression. If the length of the string exceeds eight, only the first eight characters are used.

volname is a one to six character string consisting of the letters A through Z, @, #, \$, or 0 through 9 that identifies a VS volume. This can be specified as a constant, a variable, a partial backward reference, or any other string expression. If the length of the string exceeds six, only the first six characters are used.

6. If libname is not specified, OUTLIB is used. If volname is not specified, OUTVOL is used.
7. (label) can be used as a backward reference instead of specifying the file name and location. The label must reference a DISPLAY or ENTER clause of a RUN statement that has been executed.

BOTH FORMATS

8. If the receiving string-variable is too short to contain the data, the data is truncated from the right.
9. If the receiving string-variable is longer than the data, the variable is blank-filled on the right.
10. label: is a string of one to eight alphanumeric characters that must begin with an alphabetic character and be immediately followed, with no intervening spaces, by a colon. The label can be the target of a GOTO statement. The EXTRACT statement does not assign a return code to the label.

Example

Procedure to determine the size of a user's logon procedure

```
declare &id      string(3)
declare &sysvol  string(8)
declare &size    integer

extract &id = userid, &sysvol = sysvol
extract &size=records used by &id(1,*) !! "LOGON" in LOGON on &sysvol
return code = &size
```

12.7 GOTO

Function

The GOTO statement specifies the next Procedure Language statement to be executed, and performs a branch to the labeled statement that is specified.

General Format

[label:]... GOTO target-label

General Rules

1. The GOTO statement performs a branch to the statement identified by the label target-label. The label identifies any statement. It may not refer to a DISPLAY or ENTER clause.
2. The same label can be used on more than one statement. The first occurrence of the specified label following the GOTO statement is the target. If the specified label does not follow the statement, the target is the closest textual occurrence preceding the GOTO statement. An error occurs if the specified label does not exist.
3. label: is a string of one to eight alphanumeric characters that must begin with an alphabetic character and be immediately followed, with no intervening spaces, by a colon. The label can be the target of a GOTO statement. The GOTO statement does not assign a return code to the label.

Example

```
000100 PROC
000200      GOTO A
000300 B:
000400 A: GOTO A
000500 B: RETURN
000600 A: GOTO B
```

Statements in the above procedure will be executed in the following order: 200, 400, 600, 500.

12.8 IF

Function

The IF statement controls conditional execution of procedure statements.

General Format

Format 1

[label:]... IF relational-expression [THEN] [label:]... any-statement

Format 2

[label:]... IF [NOT] EXISTS $\left\{ \begin{array}{l} \text{FILE} \quad \left\{ \begin{array}{l} \text{filename} \text{ IN libname ON volname} \\ \text{(label)} \end{array} \right\} \\ \text{LIBRARY} \left\{ \begin{array}{l} \text{libname} \text{ ON volname} \\ \text{(label)} \end{array} \right\} \\ \text{VOLUME} \left\{ \begin{array}{l} \text{volname} \\ \text{(label)} \end{array} \right\} \end{array} \right\}$
[THEN] [label:]... any-statement

General Rules

FORMAT 1

1. If the relational-expression is true, the statement within the THEN clause is executed. Otherwise, the next statement in the procedure is executed.

FORMAT 2

2. filename is a one to eight character string consisting of the letters A through Z, @, #, \$, or 0 through 9 that identifies a VS filename. This can be specified by a constant, a variable, a partial backward reference, or any other string expression. If the length of the string exceeds eight, only the first eight characters are used.

libname is a one to eight character string consisting of the letters A through Z, @, #, \$, or 0 through 9 that identifies a VS library. This can be specified by a constant, a variable, a partial backward reference, or any other string expression. If the length of the string exceeds eight, only the first eight characters are used.

volname is a one to six character string consisting of the letters A through Z, @, #, \$, or 0 through 9 that identifies a VS volume. This can be specified as a constant, a variable, a partial backward reference, or any other string expression. If the length of the string exceeds six, only the first six characters are used.

3. (label) can be used as a backward reference instead of specifying the file, and/or library, and/or volume name. The label must reference a DISPLAY or ENTER clause of a RUN statement that has been executed.
4. If FILE is specified, and if the specified file exists, the statement within the THEN clause is executed. Otherwise, the next statement in the procedure is executed.

If NOT is specified, and if the specified file does not exist, the statement within the THEN clause is executed. Otherwise, the next statement in the procedure is executed.

5. If LIBRARY is specified, and if the specified library exists, the statement within the THEN clause is executed. Otherwise, the next statement in the procedure is executed.

If NOT is specified, and if the specified library does not exist, the statement within the THEN clause is executed. Otherwise, the next statement in the procedure is executed.

6. If VOLUME is specified, and if the specified volume exists, the statement within the THEN clause is executed. Otherwise, the next statement in the procedure is executed.

If NOT is specified, and if the specified volume does not exist, the statement within the THEN clause is executed. Otherwise, the next statement in the procedure is executed.

7. A file, library, or volume does not exist if:

- The specified volume is being used exclusively by another user.
- There is insufficient buffer space to perform the IF EXISTS operation.
- There is a VTOC error.
- A disk I/O error occurs.
- The specified volume is not mounted.
- The specified file and/or library is not on the specified volume.

BOTH FORMATS

8. If the condition is false, the statement within the THEN clause is examined for errors, even though it is not executed.
9. Unquoted strings are interpreted as labels.
10. label: is a string of one to eight alphanumeric characters that must begin with an alphabetic character and be immediately followed, with no intervening spaces, by a colon. The label can be the target of a GOTO statement. The IF statement does not assign a return code to the label or labels which immediately precede IF.

Example

Procedure to simulate If-Then-Else with the Goto statement

Declare &rtncode integer

Step1: If Exists volume WORKVOL Goto TruePart

FalsePart:

Set progvol = SYSTEM

Message center 'Running tests on SYSTEM'

Goto Step2

TruePart:

Set progvol = WORKVOL

Message center 'Running tests on WORKVOL'

Step2: Run TEST1 in TESTLIB

Step3: Run TEST2 in TESTLIB

Step4: Run TEST3 in TESTLIB

Assign &rtncode = Step1 + Step2 + Step3

If &rtncode = 0

then Prompt center 'All tests passed'

Return code = &rtncode

12.9 LOGOFF

Function

The LOGOFF statement terminates all of your currently executing programs and procedures, closes all files, and automatically issues the Command Processor LOGOFF command.

General Format

[label:]... LOGOFF

General Rules

1. All currently executing programs and procedures are terminated, all files are closed, and the Command Processor LOGOFF command is executed.
2. If the procedure which contains the LOGOFF being executed is run from the EDITOR or a menu, control returns to the EDITOR or menu.
3. label: is a string of one to eight alphanumeric characters that must begin with an alphabetic character and be immediately followed, with no intervening spaces, by a colon. The label can be the target of a GOTO statement. The LOGOFF statement does not assign a return code to the label.

Example

```
PROCEDURE TO LOGOFF USER  
LOGOFF
```

12.10 MESSAGE

Function

The **MESSAGE** statement displays text on the workstation screen.

General Format

[label:]... **MESSAGE** [option] [,option]...

[ROW integer-expression] [message-line [;message-line] ...]

where:

$$\text{option} = \left\{ \begin{array}{l} \text{ALARM} = \left\{ \begin{array}{l} \text{YES} \\ \underline{\text{NO}} \end{array} \right\} \\ \text{ERASE} = \left\{ \begin{array}{l} \underline{\text{YES}} \\ \text{NO} \end{array} \right\} \end{array} \right\}$$

where:

message-line = [CENTER] [field [,field] ...]

and:

field= [attribute] ... expression

Syntax Rules

1. **ALARM** and **ERASE** can be specified in either order, but each can be specified only once.
2. The following attributes can be used:

BLANK **BLINK** **BRIGHT** **DIM** **LINE**

Refer to Section 9.1.3 for the definitions of these attributes.

General Rules

1. The **MESSAGE** statement displays text defined by the message-lines on the workstation screen as defined by the following rules. Upon display, the procedure continues executing.
2. If **ALARM** = **YES**, the workstation bell will ring when the message is displayed. If **ALARM** = **NO**, or **ALARM** is not specified, the bell will not ring.

3. If ERASE = NO, the text of the message will be written directly to the screen without first erasing the current contents of the screen. If ERASE = YES or the ERASE option is not specified, the screen will be erased entirely just prior to the message display.
4. The value for ALARM or ERASE can also be specified as a constant, a variable, a partial backward reference, or any other string expression which evaluates to YES or NO.
5. If ROW is specified, the integer expression must have a value between 1 and 24. This value indicates the row on which the message will begin. If the number of displayed lines plus the value of the ROW option exceeds 25, an error occurs.

If ROW is not specified, lines are centered vertically on the workstation screen.

6. A message can contain up to 24 lines of 79 characters. Lines longer than 79 characters are truncated from the right. If a message is longer than 24 lines, an error occurs.
7. If CENTER is not specified for a message line, that line is left justified beginning in column 2. If CENTER is specified, the text on that line is horizontally centered.
8. A semicolon causes positioning to the next line. Therefore, semicolons can be used to generate blank lines.
9. Attributes apply only to their associated field.

If conflicting attributes are specified (for example, BRIGHT and DIM in the same field), the last specified attribute applies. If compatible attributes are specified (for example, BRIGHT and BLINK), all compatible attributes apply.

When attributes are specified for at least one of two contiguous fields, one space is introduced between them. When two contiguous fields are specified, and neither has attributes, no space is introduced between them.

10. If a MESSAGE is issued and the workstation is not available, or if the procedure is running in the background, then the MESSAGE is not displayed and execution of the procedure continues.
11. Strings must be quoted. Label references cannot be used.
12. label: is a string of one to eight alphanumeric characters that must begin with an alphabetic character and be immediately followed, with no intervening spaces, by a colon. The label can be the target of a GOTO statement. The MESSAGE statement does not assign a return code to the label.

Example

Procedure RUNJOBS (runs a series of programs)

```
message erase=yes row 8 center 'JOB1 is running  '
run JOB1
message erase=no  row 8 center 'JOB1 has completed';
               center 'JOB2 is running  '
run JOB2
message erase=no  row 9 center 'JOB2 has completed';
               center 'JOB3 is running  '
run JOB3
return
```

When this procedure is run, the Procedure Interpreter displays the message "Procedure RUNJOBS in progress". When the first message statement is executed, the screen is erased and the message "JOB1 is running" is displayed on row 8. After JOB1 completes, the message "JOB1 has completed" replaces the message on row 8 and the message "JOB2 is running" is placed on row 9.

While JOB3 is running the workstation will contain the following centered messages beginning on row 8:

```
JOB1 has completed
JOB2 has completed
JOB3 is running
```

12.11 MOUNT

Function

The MOUNT statement logically mounts a disk or tape volume.

General Format

$$[\text{label:}] \dots \text{MOUNT} \begin{bmatrix} \text{DISK} \\ \text{TAPE} \end{bmatrix} \left\{ \begin{array}{l} \text{volname} \\ \text{(label)} \end{array} \right\} \text{ON unit\#} \left[\begin{array}{l} [\text{WITH}] \left\{ \begin{array}{l} \text{STANDARD} \\ \text{IBM} \\ \text{ANSI} \\ \text{NO} \end{array} \right\} \left\{ \begin{array}{l} \text{LABEL} \\ \text{LABELS} \end{array} \right\} \\ \\ \left[\begin{array}{l} [\text{FOR}] \left\{ \begin{array}{l} \text{SHARED} \\ \text{EXCLUSIVE} \\ \text{PROTECTED} \\ \text{RESTRICTED} \end{array} \right\} \left[\begin{array}{l} \text{USAGE} \\ \text{[REMOVAL]} \end{array} \right] \end{array} \right] \end{array} \right]$$

General Rules

1. The MOUNT statement logically mounts a disk or tape volume specified by volname on the unit specified by unit#, naming the type of label present and the usage restrictions for the volume.
2. The volume type (DISK or TAPE) is not optional for disk volumes named DISK or tape volumes named TAPE. For example, a disk volume named DISK is mounted by using the statement MOUNT DISK DISK ON unit#.
3. volname is a one to six character string consisting of the letters A through Z, @, #, \$, or 0 through 9 that identifies a VS volume. This can be specified as a constant, a variable, a partial backward reference, or any other string expression. If the length of the string exceeds six, only the first six characters are used.
4. (label) can be used as a backward reference instead of specifying the volume name. The label must reference a DISPLAY or ENTER clause of a RUN statement that has been executed.
5. unit# designates the number of the device on which the volume is to be mounted. unit# can be specified as a constant, a variable, a partial backward reference, or any other integer expression which evaluates to a value in the range of 1 to 99.
6. The label clause specifies the type of label present on the volume being mounted. When mounting a disk, valid options for label type are either STANDARD or NO. When mounting a tape, valid options for label type are either IBM, ANSI or NO.

When the label clause is not specified, the default label is STANDARD for disk volumes and ANSI for tape volumes.

7. The usage clause specifies the usage restrictions for the volume being mounted. When mounting a disk, valid options for usage type are SHARED, EXCLUSIVE, PROTECTED, or RESTRICTED. When mounting a tape, valid options for usage type are either SHARED or EXCLUSIVE.

When the usage clause is not specified, the default usage is SHARED for both disk and tape volumes.

A description of usage types follows:

<u>Value</u>	<u>Meaning</u>
SHARED	Any user can use or dismount the volume.
EXCLUSIVE	The user who mounts the volume is the only one who can use or dismount it.
PROTECTED	Any user can read the volume, but only the user who mounted it can modify it or dismount it.
RESTRICTED	Any user can read or modify the volume, but only the user who mounted it can dismount it.

8. A MOUNT statement executed from a procedure run in the background cannot be used to mount the removable volume of a disk unit which contains both a fixed and a removable volume (such as a Phoenix unit) where the fixed volume is the system volume.
9. Unlike the mount command issued interactively through the Command Processor, the MOUNT statement does not generate an error display if the specified volume cannot be mounted.
10. Unquoted strings are interpreted as upshifted strings. For example, Label is interpreted as "LABEL".
11. label: is a string of one to eight alphanumeric characters that must begin with an alphabetic character and be immediately followed, with no intervening spaces, by a colon. The label can be the target of a GOTO statement. The MOUNT statement assigns a return code to the label. Refer to Appendix A of this manual for the return code values.

Example

```
PROCEDURE to mount a tape and a disk
MOUNT tape VOL1 on 28 with NO LABEL for EXCLUSIVE USAGE
MOUNT SYSTEM on 16 SHARED USAGE
```


12.12 OPTIONS

Function

The **OPTIONS** statement allows specification of various Procedure Interpreter options.

General Format

$$[\text{label:}] \dots \text{OPTIONS PROCMSG} = \begin{cases} \text{YES} \\ \text{NO} \end{cases}$$

General Rules

1. In the current version of the Procedure Interpreter, the **OPTIONS** statement allows specification of one option, **PROCMSG**.

If **PROCMSG** = **NO**, display of the "Procedure in progress" message is globally disabled. If **PROCMSG** = **YES**, display of the message is globally enabled, and the message is immediately displayed.

2. The value of **PROCMSG** can also be specified as a constant, a variable, a partial backward reference, or any other string expression which evaluates to **YES** or **NO**.
3. Unquoted strings are interpreted as upshifted strings. For example, **Yes** is interpreted as **"YES"**.
4. **label:** is a string of one to eight alphanumeric characters that must begin with an alphabetic character and be immediately followed, with no intervening spaces, by a colon. The label can be the target of a **GOTO** statement. The **OPTIONS** statement does not assign a return code to the label.

Example

procedure

- * Run three procedures. **PROC1** and **PROC2** will not display
- * a "Procedure in progress" message, but **PROC3** will.

```
OPTIONS PROCMSG=NO
RUN PROC1
RUN PROC2
OPTIONS PROCMSG=YES
RUN PROC3
```

12.13 PRINT

Function

The PRINT statement places a print file in the Print Queue.

General Format

```
[label]... PRINT {filename [IN libname] [ON volname]}  
                { (label) }  
  
[, CLASS = prtclass] [ , STATUS= { SPOOL }  
                        { HOLD } ]  
  
[, FORM# = integer-expression] [, COPIES = integer-expression]  
  
[ { DISPOSITION } = { SCRATCH }  
  { DISP          }   { REQUEUE }  
                      { SAVE    } ]
```

Syntax Rules

1. CLASS, STATUS, FORM#, COPIES, and DISPOSITION can be specified in any order, but each may be specified only once.

General Rules

1. The PRINT statement places the file identified by filename IN libname ON volname in the print queue with the specified options.
2. filename is a one to eight character string consisting of the letters A through Z, @, #, \$, or 0 through 9 that identifies a VS filename. This can be specified by a constant, a variable, a partial backward reference, or any other string expression. If the length of the string exceeds eight, only the first eight characters are used.

libname is a one to eight character string consisting of the letters A through Z, @, #, \$, or 0 through 9 that identifies a VS library. This can be specified by a constant, a variable, a partial backward reference, or any other string expression. If the length of the string exceeds eight, only the first eight characters are used.

volname is a one to six character string consisting of the letters A through Z, @, #, \$, or 0 through 9 that identifies a VS volume. This can be specified as a constant, a variable, a partial backward reference, or any other string expression. If the length of the string exceeds six, only the first six characters are used.

3. If no library name for the print file is specified, SPOOLIB is used. If no volume name is specified, SPOOLVOL is used.
4. (label) can be used as a backward reference instead of specifying the file name. The label must reference a DISPLAY or ENTER clause of a RUN statement that has been executed.
5. CLASS specifies the class to which the print request is to be assigned. This value must be A through Z. It can be specified as a constant, a variable, a partial backward reference, or any other string expression.

If CLASS is not specified, the value of PRTCLASS from the Set Usage Constants function from the Command Processor is used.

6. STATUS specifies when the file will be printed. SPOOL indicates that the file will be printed as soon as the designated printer is available. HOLD places the file on the print queue, but it does not print until you release it.
7. FORM# specifies the printer form number. The value of the integer expression must be in the range 0 to 254.

If FORM# is not specified, the value of FORM# from the Set Usage Constants function from the Command Processor is used.

8. COPIES specifies the number of copies to be printed. The value of the integer expression must be in the range 1 to 32767.

If COPIES is not specified, one copy is printed.

9. DISPOSITION specifies what happens to the file after it is printed. SCRATCH deletes the file from the library; REQUEUE places the file at the bottom of the print queue; SAVE retains the file within the library, but does not requeue. The default is SCRATCH.

10. Unquoted strings are interpreted as upshifted strings. For example, Label is interpreted as "LABEL".

11. label: is a string of one to eight alphanumeric characters that must begin with an alphabetic character and be immediately followed, with no intervening spaces, by a colon. The label can be the target of a GOTO statement. The PRINT statement assigns a return code to the label. Refer to Appendix A of this manual for the return code values.

Example

Proc

Print Report in #GSPRT, Class = A, Copies = 3

12.14 PROCEDURE

Function

The PROCEDURE statement defines a procedure.

General Format

$$\left. \begin{array}{l} \text{PROCEDURE} \\ \text{PROC} \end{array} \right\} [\text{comment}]$$

Syntax Rules

1. A procedure must contain one and only one procedure statement, and it must be the first statement of the procedure.
2. Comment and blank lines may precede the PROCEDURE statement.

General Rules

1. Anything entered on the same line following PROCEDURE or PROC is ignored by the interpreter and can be used for a comment.

Example

* This is a comment line followed by a blank line

```
PROCEDURE to return a return code  
RETURN CODE = 20
```

This simple procedure returns a return code of 20. Note that the comment line and the blank line that precede the PROCEDURE statement and the text following the word PROCEDURE are ignored.

12.15 PROMPT

Function

The **PROMPT** statement displays variables and constants on the workstation and also allows the user to accept data for variables.

General Format

[label:]... **PROMPT** [option [, option]...

[ROW integer-expression] [message-line [;message-line] ...]

where:

$$\text{option} = \left\{ \begin{array}{l} \text{PFKEY} = \text{variable} \\ \text{CURROW} = \text{variable} \\ \text{CURCOL} = \text{variable} \\ \text{ALARM} = \left\{ \begin{array}{l} \text{YES} \\ \text{NO} \end{array} \right\} \\ \text{ERASE} = \left\{ \begin{array}{l} \text{YES} \\ \text{NO} \end{array} \right\} \end{array} \right\}$$

where:

message-line = [CENTER] [field [,field] ...]

and:

field= [attribute] ... expression

Syntax Rules

1. **PFKEY**, **CURROW**, **CURCOL**, **ALARM**, **ERASE** may be specified in any order, but each can be specified only once.
2. The variable associated with **PFKEY**, **CURROW**, or **CURCOL** can be an integer variable or a string variable.
3. The following attributes can be used:

BLANK	BLINK	BRIGHT	DIM	LINE
NUMERIC	TAB	UPLOW	UPPER	

Refer to Sections 9.1.3 and 9.2.1 for the definitions of these attributes.

General Rules

1. The PROMPT statement displays text defined by the message-lines on the workstation screen, allows the user to enter data for variables, and completes when the user presses a PF key or the ENTER key. The following rules further define the operation of the statement.
2. If PFKEY is specified, the associated variable will receive the PF key number (1 to 32; or 0 if ENTER is pressed) at the time the ENTER or a PF key is pressed.
3. IF CURROW is specified, the associated variable will receive the row position of the cursor (1 to 24) at the time ENTER or a PF key is pressed.
4. If CURCOL is specified, the associated variable will receive the column position of the cursor (1 to 80) at the time ENTER or a PF key is pressed.
5. If ALARM = YES, the workstation bell will ring when the message is displayed. If ALARM = NO, or is not specified, the bell will not ring.
6. If ERASE = NO, the text of the message will be written directly to the screen without first erasing the current contents of the screen. If ERASE = YES or is not specified, the screen will be erased entirely just prior to the message display.
7. The value for ALARM or ERASE can also be specified as a constant, a variable, a partial backward reference, or any other string expression which evaluates to YES or NO.
8. If ROW is specified, the integer expression must have a value between 1 and 24. This value indicates the row on which the message will begin. If the number of displayed lines plus the value of the ROW option exceeds 25, an error occurs.

If ROW is not specified, lines are centered vertically on the workstation screen.
9. A message can contain up to 24 lines of 79 characters. Lines longer than 79 characters are truncated from the right. If a message is longer than 24 lines, an error occurs.
10. If CENTER is not specified for a message line, that line is left justified beginning in column 2. If CENTER is specified, the text on that line is horizontally centered.
11. A semicolon causes positioning to the next line. Therefore, semicolons can be used to generate blank lines.
12. Attributes apply only to their associated field.

If conflicting attributes are specified (for example, BRIGHT and DIM in the same field), the last specified attribute applies. If compatible attributes are specified (for example, BRIGHT and BLINK), all compatible attributes apply.

When attributes are specified for at least one of two contiguous fields, one space is introduced between them. When two contiguous fields are specified, and neither has attributes, no space is introduced between them.

If TAB is specified as an attribute, no other attributes can be specified.

To make a field modifiable, you must specify the attributes UPPER, UPLOW, or NUMERIC. You cannot use UPPER and UPLOW with an integer-variable.

13. When an integer variable is used for a modifiable field, the displayed length of the field is eleven. Data can be entered with leading or trailing signs, but there can be no space between sign and number. The value must be between -2147483647 and 2147483647.

If the same modifiable field appears more than once in a single prompt, then the value supplied at the workstation to the first field (the uppermost, leftmost, duplicate, and modifiable field on the screen) is used for the assignment.

14. There is no relationship between PROMPT and GETPARM. Therefore, a procedure writer cannot satisfy a PROMPT from a procedure.
15. If a PROMPT is issued and the workstation is not available, or if the procedure is running in the background, an error is issued, and the procedure is canceled.
16. Strings must be quoted. Label references cannot be used.
17. label: is a string of one to eight alphanumeric characters that must begin with an alphabetic character and be immediately followed, with no intervening spaces, by a colon. The label can be the target of a GOTO statement. The PROMPT statement does not assign a return code to the label.

Example

```
Procedure
Declare &pf integer
Prompt Pfkey = &pf Row 5
    center "Press a pfkey to select one of the following options:";;
    center '( 1)',          " Run COPY          ";;
    center '( 8)',          " Run DISPLAY       ";;
    center '(16)', Bright   "Exit this procedure"
If &pf = 1 Run COPY
If &pf = 8 Run DISPLAY
If &pf = 16 Return
```

When the PROMPT statement executes, the user is presented with a menu. If PF key 1 is pressed, the COPY program runs. If PF key 8 is pressed, the DISPLAY program runs. If PF key 16, any other PF key, or ENTER is pressed, the procedure terminates.

12.16 PROTECT

Function

The PROTECT statement modifies the owner, retention period, or protection class of a disk file or library.

General Format

Format 1

```
[label]... PROTECT {filename [IN libname] [ON volname]}
                  {(label)}

TO option [,option]...
```

Format 2

```
[label]... PROTECT LIBRARY {libname [ON volname]}
                          {(label)}

TO option [,option]...
```

where:

```
option = { OWNER = string-expression
          { PERIOD = string-expression
          { FILECLAS= fileclass
```

Syntax Rules

1. OWNER, PERIOD, and FILECLAS can be specified in any order, but each may be specified only once.

General Rules

FORMAT 1

1. Format 1 is used to modify the owner, the retention period, or the protection class of a disk file.
2. If libname is not specified, OUTLIB is used.

FORMAT 2

3. Format 2 is used to modify the owner, the retention period, or the protection class of all files within the specified library.

BOTH FORMATS

4. **filename** is a one to eight character string consisting of the letters A through Z, @, #, \$, or 0 through 9 that identifies a VS filename. This can be specified by a constant, a variable, a partial backward reference, or any other string expression. If the length of the string exceeds eight, only the first eight characters are used.

libname is a one to eight character string consisting of the letters A through Z, @, #, \$, or 0 through 9 that identifies a VS library. This can be specified by a constant, a variable, a partial backward reference, or any other string expression. If the length of the string exceeds eight, only the first eight characters are used.

volname is a one to six character string consisting of the letters A through Z, @, #, \$, or 0 through 9 that identifies a VS volume. This can be specified as a constant, a variable, a partial backward reference, or any other string expression. If the length of the string exceeds six, only the first six characters are used.

5. If **volname** is not specified, **OUTVOL** is used.
6. **(label)** can be used as a backward reference instead of specifying the file and/or library name and volume. The label must reference a **DISPLAY** or **ENTER** clause of a **RUN** statement that has been executed.
7. **OWNER** is used to modify the file owner. The value of the string expression must be a three-character user id.
8. **PERIOD** is used to modify the retention period, the length of time that the file remains protected on the system. The value of the string expression is specified in days, and must be in the range 0 to 999.
9. **FILECLAS** is used to modify the file protection class. The value of fileclass must be A through Z, #, \$, @ or blank. It can be specified as a constant, a variable, a partial backward reference, or any other string expression.
10. Unlike the **PROTECT** option issued interactively through the Command Processor, the **PROTECT** statement does not generate an error message if the specified options cannot be modified.
11. Unquoted strings are interpreted as upshifted strings. For example **Label** is interpreted as **"LABEL"**.

12. label: is a string of one to eight alphanumeric characters that must begin with an alphabetic character and be immediately followed, with no intervening spaces, by a colon. The label can be the target of a GOTO statement. The PROTECT statement assigns a return code to the label. Refer to Appendix A of this manual for the return code values.

Example

Procedure

Protect EXPENSES in ACCTLIB on VOL444 to OWNER = JR, FILECLAS = A

Protect Library SALARIES to OWNER = MGR, PERIOD = 365

12.17 RENAME

Function

The **RENAME** statement renames a disk file and/or library.

General Format

Format 1

```
[label]...  RENAME  { filename [ IN libname] [ ON volname] }  
                  { (label) }  
  
                TO filename [ IN libname]
```

Format 2

```
[label:]... RENAME  LIBRARY { libname [ON volname] }  
                           { (label) }  
  
                TO libname
```

General Rules

FORMAT 1

1. Format 1 renames the first filename specified to the filename specified after TO.
2. If no library is specified, OUTLIB is used. If no volume is specified, OUTVOL is used.

FORMAT 2

3. Format 2 renames the first library specified to the library specified after TO.

BOTH FORMATS

4. **filename** is a one to eight character string consisting of the letters A through Z, @, #, \$, or 0 through 9 that identifies a VS filename. This can be specified by a constant, a variable, a partial backward reference, or any other string expression. If the length of the string exceeds eight, only the first eight characters are used.

libname is a one to eight character string consisting of the letters A through Z, @, #, \$, or 0 through 9 that identifies a VS library. This can be specified by a constant, a variable, a partial backward reference, or any other string expression. If the length of the string exceeds eight, only the first eight characters are used.

volname is a one to six character string consisting of the letters A through Z, @, #, \$, or 0 through 9 that identifies a VS volume. This can be specified as a constant, a variable, a partial backward reference, or any other string expression. If the length of the string exceeds six, only the first six characters are used.

5. (label) can be used as a backward reference instead of specifying the file or library name. The label must reference a DISPLAY or ENTER clause of a RUN statement that has been executed.
6. Unlike the RENAME command issued interactively through the Command Processor, the RENAME statement does not generate an error message if the specified file or library cannot be renamed.
7. Unquoted strings are interpreted as upshifted strings. For example, Label is interpreted as "LABEL".
8. **label:** is a string of one to eight alphanumeric characters that must begin with an alphabetic character and be immediately followed, with no intervening spaces, by a colon. The label can be the target of a GOTO statement. The RENAME statement assigns a return code to the label. Refer to Appendix A of this manual for the return code values.

Example

```
PROC
RENAME FILE1 IN TESTLIB TO FILE2
RENAME LIBRARY TESTLIB TO TESTS
RENAME PROCA IN PROCLIB ON SYS356 TO PROCA IN TESTS
```

12.18 RETURN

Function

The RETURN statement unconditionally terminates procedure execution.

General Format

[label:]...RETURN [CODE = integer-expression]

General Rules

1. The RETURN statement unconditionally terminates procedure execution, and passes a return code and returns control to the invoking routine.
2. If CODE is specified, the value of the integer expression is returned as the procedure return code.

If CODE is not specified, zero is returned as the procedure return code.

3. If a procedure does not contain a return statement, it will terminate when the last statement in the procedure file is executed. The procedure return code will be zero.
4. Unquoted strings are interpreted as labels.
5. label: is a string of one to eight alphanumeric characters that must begin with an alphabetic character and be immediately followed, with no intervening spaces, by a colon. The label may be the target of a GOTO statement. The RETURN statement does not assign a return code to the label.

Example

Procedure

Label: Run DEMO

Return code = label + 1000

12.19 RUN

Function

The RUN statement runs a program or another procedure.

General Format

```
[label:]... RUN {filename [IN libname] [ON volname]}
                  (label)

[ USING {variable } [ , {variable } ] ... ]
  {expression} {expression}

[ERROR EXIT [IS] label] [CANCEL EXIT [IS] label]

[[label:] DISPLAY {prname } [keyword = value [, keyword = value]...] ]...
  {inner-label}      (label)

[[label:] ENTER {prname } [pfkey ,]
  {inner-label}

[keyword = value [,keyword = value]...] ] ...
 (label)
```

Syntax Rules

1. The CANCEL EXIT and ERROR EXIT clauses can be specified in any order, but each can be specified only once.
2. The DISPLAY and ENTER clauses can be specified multiple times in any order.
3. A keyword in a DISPLAY or ENTER clause can contain hyphens.

General Rules

1. The RUN statement runs the program or procedure specified by filename.
2. filename is a one to eight character string consisting of the letters A through Z, @, #, \$, or 0 through 9 that identifies a VS filename. This can be specified by a constant, a variable, a partial backward reference, or any other string expression. If the length of the string exceeds eight, only the first eight characters are used.

libname is a one to eight character string consisting of the letters A through Z, @, #, \$, or 0 through 9 that identifies a VS library. This can be specified by a constant, a variable, a partial backward reference, or any other string expression. If the length of the string exceeds eight, only the first eight characters are used.

volname is a one to six character string consisting of the letters A through Z, @, #, \$, or 0 through 9 that identifies a VS volume. This can be specified as a constant, a variable, a partial backward reference, or any other string expression. If the length of the string exceeds six, only the first six characters are used.

3. The program or procedure can be specified in the following ways:
 - the filename, library, and volume are specified
 - the filename and library are specified
 - the filename and volume are specified
 - the filename alone is specified.

If the filename, library, and volume are all specified, the system searches the library and volume for the file. If the library does not exist on the volume, the file location must be respecified. If the library exists on the volume, but the file is not found within it, the system searches SYSLIB on SYSVOL. If the file is not found there, the file location must be respecified.

If only the filename and library are specified, the system searches for the file in the specified library on PROGVOL. If the file is not found, the system searches for the file in the specified library on SYSVOL. If the specified library does not exist on SYSVOL, the file location must be respecified. If the specified library exists on SYSVOL, but the file is not found within it, the system searches SYSLIB on SYSVOL. If the file is not found there, the file location must be respecified.

If only the filename and volume are specified, the system searches for the file in PROGLIB on the specified volume. If the file is not found, the system searches for the file in SYSLIB on the specified volume. If SYSLIB does not exist on the specified volume, the file location must be respecified. If SYSLIB exists on the specified volume, but the file is not found within it, the system searches SYSLIB on SYSVOL. If the file is not found there, the file location must be respecified.

If only the filename is specified, the system searches for the file in PROGLIB on PROGVOL. If the file is not found, the system searches SYSLIB on SYSVOL. If the file is not found there, the file location must be respecified.

4. (label) can be used as a backward reference instead of specifying the file name and location. The label must reference a DISPLAY or ENTER clause of a RUN statement that has been executed.

5. The USING clause specifies the parameters to be passed. Refer to section 7.3 for a discussion of parameter passing.

If a boolean expression is specified as a parameter in the USING clause, the value of the expression is passed as an integer expression having a value of 0 or 1.

6. If ERROR EXIT is specified, and the program or procedure cannot be run, control will be transferred to the specified label. The label of the RUN statement will receive a return code, indicating why the run failed. Refer to Appendix A of this manual for the return code values.

If ERROR EXIT is not specified, and the program or procedure cannot be run, a GETPARM requests respecification of the file and its location.

7. If CANCEL EXIT is specified, and the program or procedure run is canceled, control will be transferred to the specified label. The label of the RUN statement will not receive a return code.

If CANCEL EXIT is not specified, and the program or procedure run is canceled, control will not be transferred back to the procedure containing this RUN statement.

8. The DISPLAY and ENTER clauses are used to satisfy GETPARMs in the program or procedure being run. Refer to Sections 3.3 and 3.5 for a discussion of DISPLAY and ENTER.

pfkey is the PF key that is used to satisfy a GETPARM. pfkey can be specified as a constant, a variable, a partial backward reference, or any other integer expression which evaluates to a value between 1 to 32.

keyword is a keyword of a GETPARM.

prname is the prname of a GETPARM.

inner-label is the label of a nested procedure.

9. Prior to linking to the program or procedure to be run, the Procedure Interpreter saves the current contents of the workstation screen. After the RUN statement has been executed, the saved screen is redisplayed on the workstation, completely replacing anything that was displayed by the program or procedure run.
10. Strings must be quoted when used in the USING clause. Unquoted strings are interpreted as labels when used in the CANCEL EXIT and ERROR EXIT clauses. Otherwise, unquoted strings are interpreted as upshifted strings. For example Label is interpreted as "LABEL".

11. label: is a string of one to eight alphanumeric characters that must begin with an alphabetic character and be immediately followed, with no intervening spaces, by a colon. The label associated with the RUN statement can be the target of a GOTO statement. The label associated with the DISPLAY or ENTER clauses cannot be the target of a GOTO statement, but is used for backward referencing.

The RUN statement assigns a return code to the label which immediately precedes RUN. The return code from the RUN statement is the return code from the program or procedure that was run.

Example

Procedure

```
Declare &Parm1 integer
Run Demo in Demolib on Vol444 Using &Parm1, "DEMO"
    Error Exit is Error      Cancel Exit is Cancel
    Display Input Key1 = Test
    Enter EOJ 16, Option = A
```

Return code = 100

Error: Return code = 86

Cancel: Return Code = 93

12.20 SCRATCH

Function

The SCRATCH statement scratches a disk file or library.

General Format

Format 1

```
[label:]... SCRATCH {filename [IN libname] [ON volname]}  
                    {(label)}
```

Format 2

```
[label:]... SCRATCH LIBRARY {libname [ON volname]}  
                             {(label)}
```

General Rules

FORMAT 1

1. Format 1 scratches the specified file.
2. If no library is specified, OUTLIB is used.

FORMAT 2

3. Format 2 scratches the specified library.

BOTH FORMATS

4. filename is a one to eight character string consisting of the letters A through Z, @, #, \$, or 0 through 9 that identifies a VS filename. This can be specified by a constant, a variable, a partial backward reference, or any other string expression. If the length of the string exceeds eight, only the first eight characters are used.

libname is a one to eight character string consisting of the letters A through Z, @, #, \$, or 0 through 9 that identifies a VS library. This can be specified by a constant, a variable, a partial backward reference, or any other string expression. If the length of the string exceeds eight, only the first eight characters are used.

volname is a one to six character string consisting of the letters A through Z, @, #, \$, or 0 through 9 that identifies a VS volume. This can be specified as a constant, a variable, a partial backward reference, or any other string expression. If the length of the string exceeds six, only the first six characters are used.

5. If no volume is specified, OUTVOL is used.
6. (label) can be used as a backward reference instead of specifying the file or library name. The label must reference a DISPLAY or ENTER clause of a RUN statement that has been executed.
7. Unlike the SCRATCH command issued interactively through the Command Processor, the SCRATCH statement does not generate an error message if the specified file or library cannot be scratched.
8. Unquoted strings are interpreted as upshifted strings. For example Label is interpreted as "LABEL".
9. label: is a string of one to eight alphanumeric characters that must begin with an alphabetic character and be immediately followed, with no intervening spaces, by a colon. The label can be the target of a GOTO statement. The SCRATCH statement assigns a return code to the label. Refer to Appendix A of this manual for the return code values.

Example

Procedure

Scratch Junk in Junklib on VOL444
Scratch library Testlib

12.21 SET

Function

The **SET** statement sets default values for file assignments, procedure submittal defaults, and print mode options.

General Format

[label:]... **SET** keyword = expression [, keyword = expression] ...

Syntax Rules

1. A keyword identifies a field for which a default parameter value can be specified. The legal keywords for the **SET** statement are:

FILECLAS	JOBQUEUE	PROGLIB	SPOOLIB
FORM#	LINES	PROGVOL	SPOOLSYS
INLIB	OUTLIB	PRTCLAS	SPOOLSYSRC
INVOL	OUTVOL	PRTFCLAS	SPOOLVOL
JOBCLASS	PRINTER	RUNLIB	WORKVOL
JOBLIMIT	PRNTMODE	RUNVOL	

Refer to Section 5.1 for definitions of these keywords.

General Rules

1. The **SET** statement sets the specified keywords to the values of the associated expressions.
2. An integer expression is required for **FORM#**, **JOBLIMIT**, **LINES**, and **PRINTER**. A string expression is required for all other keywords.
3. The expression associated with **SPOOLSYSRC** must consist of a single variable. **SPOOLSYSRC** is a special keyword used to receive a return code when setting **SPOOLSYS**. Refer to Section 5.1.
4. All parameters that can be specified with the **SET** statement, with the exception of **PROGLIB** and **PROGVOL**, can be specified interactively through the Command Processor.
5. Unquoted strings are interpreted as upshifted strings. For example, **Label** is interpreted as **"LABEL"**.
6. **label:** is a string of one to eight alphanumeric characters that must begin with an alphabetic character and be immediately followed, with no intervening spaces, by a colon. The label can be the target of a **GOTO** statement. The **SET** statement does not assign a return code to the label.

Example

Procedure

Set inlib = GSSRCE, outlib = GSOBJ, PRNTMODE = K

12.22 SUBMIT

Function

The **SUBMIT** statement submits a program or procedure file into the Program or Procedure Queue for noninteractive execution. Selected local variables can be passed to the submitted procedure or program. All global variables can also be passed to the submitted procedure.

General Format

```
[label:]... SUBMIT {procname [IN libname] [ON volname]}  
                  {(label)}
```

```
[AS procedure-id ] [USING parameter [, parameter] ...]
```

```
[ , GLOBALS= { YES } ]      [ , ENVIRONMENT= { YES } ]  
                        { NO }      { NO }
```

```
[ , CLASS = class ] [ , STATUS = { RUN }  
                        { HOLD } ]
```

```
[ , DUMP = { PROGRAM } ]      [ , CPULIMIT = { hh:mm:ss }  
                        { YES }      { ss }  
                        { NO }
```

```
[ , ACTION = { CANCEL } ] [ , { DISPOSITION } = REQUEUE  
                        { WARN }  
                        { PAUSE } ] [ , { DISP }
```

Syntax Rules

1. GLOBALS, ENVIRONMENT, CLASS, STATUS, DUMP, CPULIMIT, ACTION, and DISPOSITION can be specified in any order, but each may be specified only once.

General Rules

1. The **SUBMIT** statement submits a program or procedure file into the Program or Procedure Queue for noninteractive execution. With the **USING** clause, selected local variables can be passed to the submitted procedure or program. With the **GLOBALS** option, all global variables can also be passed to the submitted procedure.

2. filename is a one to eight character string consisting of the letters A through Z, @, #, \$, or 0 through 9 that identifies a VS filename. This can be specified by a constant, a variable, a partial backward reference, or any other string expression. If the length of the string exceeds eight, only the first eight characters are used.

libname is a one to eight character string consisting of the letters A through Z, @, #, \$, or 0 through 9 that identifies a VS library. This can be specified by a constant, a variable, a partial backward reference, or any other string expression. If the length of the string exceeds eight, only the first eight characters are used.

volname is a one to six character string consisting of the letters A through Z, @, #, \$, or 0 through 9 that identifies a VS volume. This can be specified as a constant, a variable, a partial backward reference, or any other string expression. If the length of the string exceeds six, only the first six characters are used.

3. If a filename, library, and volume are all specified, and the file does not exist, the SUBMIT is unsuccessful. If a label is associated with the SUBMIT statement, a return code is assigned to the label identifying the problem.

If no library and no volume name is specified, the file is searched for in PROGLIB on PROG VOL. If that file does not exist, the file is searched for in SYSLIB on SYSVOL. If that file does not exist, an error occurs.

If a library, but no volume name is specified, the file is searched for in the specified library on PROG VOL. If that file does not exist, the file is searched for in the specified library on SYSVOL. If that file does not exist, an error occurs.

If a volume, no library name is specified, the file is searched for in PROGLIB on the specified volume. If that file does not exist, the file is searched for in SYSLIB on the specified volume. If that file does not exist, an error occurs.

4. (label) can be used as a backward reference instead of specifying the file name and location. The label must reference a DISPLAY or ENTER clause of a RUN statement that has been executed.
5. If AS procedure-id is specified, the procedure-id is used to identify the procedure on the procedure queue.

If AS procedure-id is not specified, the system automatically assigns one.

6. If USING is specified, the listed parameters are passed by value to the submitted procedure or program.

If a boolean expression is specified as a parameter in the USING clause, the value of the expression is passed as an integer expression having a value of 0 or 1.

7. **GLOBALS** specifies whether all global variables accessible to the submitting procedure are to be made available to the submitted procedure. If **GLOBALS = YES**, global variables are made available to the submitted procedure; otherwise, they are not.
8. **ENVIRONMENT** specifies whether the user's current usage constants will apply to the submitted procedure. If **ENVIRONMENT = YES**, current usage constants will apply; otherwise, they will not.
9. **CLASS** specifies the class to which the submit request is to be assigned. This value must be A through Z. It can be specified as a constant, a variable, a partial backward reference, or any other string expression.

If **CLASS** is not specified, the value of **JOBCLASS** from the Set Usage Constants function from the Command Processor is used.

10. **STATUS** specifies the queueing status for the submitted procedure. **RUN** specifies that the file will be executed. **HOLD** places the file on the procedure queue, but it does not execute until it is released.
11. **DUMP** specifies whether a program dump is produced on abnormal termination. If **DUMP = YES**, a dump is produced for the job submitted. If **DUMP = NO**, no dump is created. If **DUMP = PROGRAM**, a dump is produced only if the program terminates abnormally.
12. **CPULIMIT** specifies the amount of CPU time that can pass before the specified or default **ACTION** occurs. You can specify the maximum CPU processing time allowed in hours, minutes, and seconds, or in seconds alone. The values specified must fall within the following ranges: hours, 0 through 99; minutes, 0 through 59; seconds, 0 through 59; seconds only, 0 through 359999. If **CPULIMIT** is not set or is set to zero, infinite time is implied.
13. **ACTION** specifies the action taken if the **CPULIMIT** is exceeded. **CANCEL** terminates procedure execution and issues an error. **WARN** issues a warning, and continues execution of the procedure. **PAUSE** halts execution and gives the option to continue or cancel.

ACTION is a valid clause only when a nonzero **CPULIMIT** is specified. If **ACTION** is specified, but the **CPULIMIT** is 0 or is not specified, an error occurs.

14. **DISPOSITION** specifies whether the procedure will be queued again after execution. **REQUEUE** is the only option, and specifies that the procedure will be requeued.

15. The value of GLOBALS, ENVIRONMENT, CLASS, STATUS, DUMP, ACTION, DISPOSITION, as well as hh, mm, and ss of CPULIMIT can also be specified as a constant, a variable, a partial backward reference or any other expression which evaluates to a valid value for the given option.
16. The Procedure Interpreter generates a procedure and submits it if the USING clause is coded, GLOBALS is equal to YES, ENVIRONMENT is equal to YES, or if a program file is being submitted. Refer to Section 5.7 for more information.
17. Unquoted strings are interpreted as upshifted strings. For example Label is interpreted as "LABEL". Strings must be quoted when used in the USING clause.
18. label: is a string of one to eight alphanumeric characters that must begin with an alphabetic character and be immediately followed, with no intervening spaces, by a colon. The label can be the target of a GOTO statement. The SUBMIT statement assigns a return code to the label. Refer to Appendix A of this manual for the return code values.

Example

Proc

```
Declare &parm2 integer
Submit  job100 in gslib on vol600 as demo using "test", &parm2,
        globals = yes, environment = yes, cpulimit = 0:01:30,
        action = warn
```

12.23 TRACE

Function

The TRACE statement initiates or terminates tracing.

General Format

Format 1

```
[label:]... TRACE  [ INTO  { file [IN library] [ON volume] }  
                    { (label) } ]  
  
                    [ , RESOURCES = { YES } ]  
                    [      { NO } ]  
  
                    [ , SCRATCH = { YES } ]  
                    [      { NO } ]  
  
                    [ , STATEMENTS = { RUN } ]  
                    [      { ALL } ]  
  
                    [ , VARIABLES = { YES } ]  
                    [      { NO } ]
```

Format 2

```
[label:]... TRACE END
```

Syntax Rules

1. RESOURCES, SCRATCH, STATEMENTS and VARIABLES can be specified in any order, but each may be specified only once.

General Rules

FORMAT 1

1. Format 1 is used to initiate tracing or to change options while tracing is in progress. To change an option while tracing is in progress, code the TRACE statement without the INTO clause, specifying the options to be changed. To change the trace file while tracing is in progress, code the TRACE statement with the INTO clause, specifying the options to be changed, if any.

2. **filename** is a one to eight character string consisting of the letters A through Z, @, #, \$, or 0 through 9 that identifies a VS filename. This can be specified by a constant, a variable, a partial backward reference, or any other string expression. If the length of the string exceeds eight, only the first eight characters are used.

libname is a one to eight character string consisting of the letters A through Z, @, #, \$, or 0 through 9 that identifies a VS library. This can be specified by a constant, a variable, a partial backward reference, or any other string expression. If the length of the string exceeds eight, only the first eight characters are used.

volname is a one to six character string consisting of the letters A through Z, @, #, \$, or 0 through 9 that identifies a VS volume. This can be specified as a constant, a variable, a partial backward reference, or any other string expression. If the length of the string exceeds six, only the first six characters are used.

3. **(label)** can be used as a backward reference instead of specifying the file name and location. The label must reference a **DISPLAY** or **ENTER** clause of a **RUN** statement that has been executed.
4. If **INTO** is specified, all trace information is written into the named file. If no library is specified, **SPOOLIB** is used. If no volume is specified, **SPOOLVOL** is used.

If **INTO** is not specified, and tracing is being initiated, all trace information is written into a file with file name the same as the current procedure, and library **SPOOLIB** and volume **SPOOLVOL**.

5. **RESOURCES** specifies whether resource usage statistics for **RUN** statements will be traced. If **RESOURCES = YES**, resource usage statistics (CPU and elapsed time) are traced; otherwise, they are not.
6. **SCRATCH** specifies whether the trace file will be scratched before tracing begins. If **SCRATCH = YES**, the trace file is scratched; otherwise it is not.
7. **STATEMENTS** specifies whether all statements or only **RUN** statements will be traced. If **STATEMENTS = RUN**, only **RUN** statements are traced; otherwise, all statements are traced.
8. **VARIABLES** specifies whether runtime values of variables will be traced. If **VARIABLES = YES**, runtime values of variables are traced; otherwise, they are not.
9. The value of **RESOURCES**, **SCRATCH**, **STATEMENTS**, or **VARIABLES** can also be specified as a constant, a variable, a partial backward reference or any other expression which evaluates to a valid value for the given option.

10. Unquoted strings are interpreted as upshifted strings. For example Label is interpreted as "LABEL".

FORMAT 2

11. Format 2 is used to terminate tracing explicitly. Tracing is implicitly terminated by terminating the procedure that initiated tracing.

BOTH FORMATS

12. label: is a string of one to eight alphanumeric characters that must begin with an alphabetic character and be immediately followed, with no intervening spaces, by a colon. The label can be the target of a GOTO statement. The TRACE statement assigns a return code to the label. Refer to Appendix A of this manual for the return code values.

Example

Procedure

Trace into tfile1 in gstrace on vol444, scratch = yes, variables = yes

.

.

.

Trace end

12.24 USING

Function

The USING statement declares the formal parameters to a procedure.

General Format

[label:]... USING variable [, variable] ...

AS {STRING (expression)}
{INTEGER}

[, variable [, variable] ... AS {STRING (expression)}
{INTEGER}] ...

Syntax Rules

1. When the USING statement is used, it must be the first statement following the PROCEDURE statement.

General Rules

1. The USING statement declares the formal parameters to a procedure, specified by the listed variables, and the attributes of the parameters.
2. The parameter type can be STRING or INTEGER. If the parameter type is STRING, it appears in the form STRING (expression), where expression is an integer expression whose value is between 0 and 256. This value defines the size of a string parameter.
3. The number of parameters passed must equal the number of parameters declared in the USING statement.
4. There is no type checking between the type of parameter passed and the type of the parameter declared in the USING statement. Therefore, it is the user's responsibility to insure consistency.
5. label is a one to eight character alphanumeric value that must begin with an alphabetic character and contain no embedded spaces. The label may be the target of a GOTO statement. The USING statement does not assign a return code to the label.

Example

Procedure SUBPROC, which illustrates a USING statement
Using &file, &library string(8), &volume string(6)

Run COBOL

enter input file= &file, library= &library, volume= &volume

enter options

enter output file= &file, library= GSOBJ, volume= LANG

PART III

Appendices

APPENDIX A

RETURN CODE VALUES- VS SYSTEM UTILITIES & PROCEDURE LANGUAGE STATEMENTS

All programs that run on the VS generate return codes which, if other than zero, can be displayed on the workstation after program completion. Some system utility programs, along with the procedure statements DISMOUNT, MOUNT, PROTECT, RENAME, SCRATCH, and SUBMIT generate return codes that indicate success or failure and can indicate the type of failure that occurred. The nonzero return codes and their meanings are listed below.

VS SYSTEM UTILITIES

<u>Program Name</u>	<u>Return Code</u>	<u>Meaning</u>
BACKUP	n	The integer n is the number of errors found.
COPY	100	No COPY took place.
	104	Some program privileges were lost.
	108	There has been a disk error. Run LISTVTOC.
	112	No space is available in the output volume.
	116	There has been an I/O error on output.
	120	A boundary violation occurred.
	124	A key was out of sequence.
	128	A duplicate key was found.
	132	The primary extent was exceeded.
	136	There was a DMS error.
	140	A duplicate file name was encountered.
	144	The COPY was completed in I/O mode after the primary extent was exceeded.

<u>Program Name</u>	<u>Return Code</u>	<u>Meaning</u>
DISKINIT	4	DISKINIT was terminated by the user.
	8	There was insufficient space in the I/O buffer.
	12	The Mount operation was unsuccessful.
	16	A bad disk sector was encountered.
	20	Bad return code by MOUNT SVC.
	24	Bad return code by FREEBUF SVC.
	28	Bad return code by XIO SVC.
	32	A disk I/O error occurred.
EZFORMAT	1-4	Warning. The program will probably run.
	5-7	Severe error. The program will not run correctly.
	8-16	Fatal error. No object code was generated.
LINKER	4	Either unsatisfied external reference(s), or labeled COMMON sections of unequal lengths were encountered (FORTRAN only).
	8	-- Multiple entry points with the same name were encountered. -- Static section in segment 0 address space was encountered. -- Initialized blank COMMON static section was encountered (FORTRAN only). -- Illegal replacement attempted (possible only in object programs generated in FORTRAN). This error occurs when a code or static section and a labeled or blank COMMON section of the same name are encountered.
LISTVTOC	4	An Extent was lost on disk.
	8	There was an error on a file label.
	12	There was an error in a library directory.
	16	There was an error in the VTOC.

SORT

- 4 No record to be sorted. Input records did not meet selection criteria, or the input file specified by the user was empty.
- 8 Insufficient space in the stack or I/O buffer.
- 12 The record size is more than 1024 bytes long.
- 16 There was an invalid sort key.
- 20 An unexpected program check occurred.
- 24 Input records in the file to be merged were out of order. The program can not proceed.

All compilers

- 1-4 Warning.
- 5-8 Severe error. The program will not execute correctly.
- 9-16 Fatal error. The program will not execute.

PROCEDURE LANGUAGE RETURN CODE VALUES

<u>Statement Name</u>	<u>Return Code</u>	<u>Meaning</u>
DECLARE	0	All variables were declared.
	4	No variables were declared.
	8	One or more variables were declared, and one or more variables were not declared.
DISMOUNT	4	Input volume name is blank, or bytes 0-1 in the input are nonzero.
	8	Volume cannot be found.
	12	Volume cannot be dismounted.
	16	Device detached.
	20	Volume in use by a user or the operating system.
	24	Volume reserved by another user.
	28	GETMEM (SVC) pool failure.
	32	Device is reserved by another task.
MOUNT	4	Successful mount, but new volume label type does not agree with the input parameters.
	8	Successful mount, but the new volume name is not the volume name requested.
	12	Disk or tape I/O error detected while reading the new volume label, or the new volume has a bad VTOC. VCBSEB is set to blank.
	16	Device is not a disk or tape, or the device number is not valid.
	20	Device detached.
	24	Volume type (REM or FIX) not found.
	28	Request to mount unlabeled volume on disk other than diskette.
	32	Input volume name is already blank.

MOUNT	36	Volume already mounted, or duplicate volume name entered.
	40	Volume is currently in use by a user or by the operating system.
	44	Volume reserved by another user for exclusive use.
	48	I/O buffer is insufficient to perform mount.
	52	Unable to allocate space for tape I/O control blocks.
	56	Invalid request. Work and/or spool filing requested in a nonlabeled volume.
	60	Invalid request. Nonstandard addressing attempted with standard label option or on a hard-sectored device.
	64	Wrong media. Soft-sectored diskette inserted into a device for hard-sectored diskettes only.
	68	Wrong media. Hard-sectored diskette inserted into a device for soft-sectored diskettes only.
	72	Wrong media. Hard-sectored diskette inserted for a nonstandard addressing request.
	76	Wrong addressing mode. Caller requested MOUNT for standard addressing but diskette is nonstandard addressing.
	80	Device reserved by another user.
	84	PF16 was entered when the mount message was displayed.
PRINT	4	Volume is not mounted.
	8	Volume used exclusively by another user.
	12	All buffers in use; unable to perform verification.
	16	Library not found.
	20	File not found.
	24	Improper file type.

PRINT	32	VTOC error.
	36	VTOC error.
PROTECT	4	Volume is not mounted.
	8	Volume used exclusively by another user.
	12	All buffers in use; no protection change.
	16	Library not found.
	20	File not found.
	24	Update access denied; no protection change.
	32	File in use; no protection change.
	36	VTOC error.
	40	VTOC error.
	44	Invalid argument list address.
	48	I/O error; VTOC unreliable.
	52	Open or protected files bypassed in protecting library.
	56	Invalid new protection data.
RENAME	4	Volume is not mounted.
	8	Volume used exclusively by another user.
	12	All buffers in use; no rename.
	16	Library not found.
	20	File not found.
	24	Update access to file protection class denied; no rename.
	32	Unexpired file; no rename.
	36	VTOC error.
	40	VTOC error.
	44	Invalid argument list address.

RENAME	48	I/O error. VTOC is unreliable.
	52	New file or library name already exists, no rename.
	56	New filename invalid (or first character #), no rename.
	60	VTOC is currently full.
	64	Reserved bits in the parameter list options byte are nonzero.

RUN (RUN return codes only apply if the ERROR EXIT is taken. Otherwise, the return code from RUN is the return code from the program or procedure that is run.)

0	Invalid program file.
4	Volume is not mounted.
8	Volume is in exclusive use by another user.
12	No buffers are available for use by the Link SVC.
16	Library cannot be found.
20	File cannot be found.
24	Maximum number of allowed link levels has been exceeded.
28	User has insufficient rights to access this file.
32	VTOC error occurred while locating file.
36	VTOC error occurred while locating file.
40	VTOC error occurred while locating file.
44	I/O error occurred while reading VTOC.
48	VTOC error occurred while locating file.
52	Invalid program file.
56	File is currently in use by another program.

SCRATCH	4	Volume is not mounted.
	8	Volume used exclusively by another user.
	12	All buffers in use. No scratch.
	16	Library not found.
	20	File not found.
	24	Update access to file protection class denied (single file scratch only).
	32	File in use. No scratch.
	36	VTOC error.
	40	VTOC error.
	44	Invalid argument list address.
	48	I/O error. VTOC is unreliable.
	52	Open, protected and/or unexpired file(s) bypassed in scratching library.
SUBMIT	4	Volume is not mounted.
	8	Volume used exclusively by another user.
	12	All buffers in use; unable to perform verification.
	16	Library not found.
	20	File not found.
	24	Improper file type.
	32	VTOC error.
	36	VTOC error.
TRACE	0	Tracing has successfully started.
	4	Tracing is already in progress. Specified options have been updated.
	8	TRACE END was executed, but tracing was not in progress.

TRACE	12	Tracing was initiated at a higher link level, and is in progress. The new TRACE statement is ignored.
	16	File format error. (The specified trace file is not a print file.)
	20	Insufficient access rights to trace file.
	24	Trace file possession conflict.
	28	Insufficient VTOC space; can not create or access trace file.
	32	Insufficient space on volume for trace file.
	36	Trace file volume is not mounted.

APPENDIX B
INCOMPATIBILITIES BETWEEN VERSION 2.7.7 AND
THE CURRENT VERSION OF THE PROCEDURE INTERPRETER

There are a small number of incompatibilities between versions 02.07.07 and the current version of the Procedure Interpreter. Though some users may be adversely affected, the benefits obtained by using the current version will outweigh any problems caused by its introduction onto an existing VS system.

The following list describes the incompatibilities:

1. Previously, keyvalues of DISPLAY and ENTER clauses could consist of any string of 1 or more characters without enclosing them in quotes. E.g. DISPLAY INPUT FILE=???.

Currently, you must enclose in quotes any keyvalues that do not consist of A-Z, a-z, @, #, \$, or 0-9. E.g. DISPLAY INPUT FILE='???'.

2. Previously, if no RETURN statement were coded in a procedure, the return code of the procedure would equal the contents of register 0. Some people used this bug to their advantage by writing procedures in which the last statement ran a utility such as a compiler. The return code of the compiler became the return code of the procedure.

Currently, the return code of a procedure is always zero unless otherwise specified by a RETURN CODE statement.

3. Previously, the prname of the input file screen was PROCFILE. Currently, the prname of the input file screen is INPUT.
4. Previously, the minimum value that an integer variable could hold was -2147483648. Currently, the minimum value for an integer variable is -2147483647.

Previously, the allowable range of an integer constant was -99999999 to 99999999. Currently, the allowable range for an integer constant is -2147483647 to 2147483647.

5. Previously, misspellings were not flagged. Currently, the following misspellings are flagged:

<u>Word</u>	<u>Statement that contains it</u>
TO	PROTECT and RENAME
CODE	RETURN
USING	USING
keywords of length 8	SET

6. Previously, the maximum number of records that a procedure file could contain had no limit. Currently, the limit is 32767.
7. Currently, only one label definition for each DISPLAY and ENTER clause of the RUN statement is allowed..

APPENDIX C

GLOSSARY

ASSIGN	Statement that assigns values to variables defined in the procedure.
background processing	Running programs or procedures requiring no user intervention, by submitting them to the operating system Procedure Queue, thus freeing the workstation for interactive use.
backward reference	The process by which a statement is referenced from another statement to obtain parameter information.
built-in function	An expression whose value is determined by accessing a predefined routine. The built-in functions are &length, &time, and &date.
comment	Any user-written message that is to be ignored by the Procedure Interpreter. The Procedure Interpreter interprets comments as blanks.
constant	A sequence of characters whose value is implied by that sequence of characters. The particular value represented may be a string or an integer.
DECLARE	Statement that defines the variables to be used within a procedure.
DESTROY	Statement that closes the currently executing procedure file, attempts to scratch it, and returns control to the invoking routine.
DISMOUNT	Statement that logically dismounts a disk or tape volume.
expression	A constant, variable, built-in function, label reference, or backward reference, appearing alone or in combination with operators, which represents a value.

EXTRACT	Statement that extracts information from the system and stores it in the variables defined in a procedure.
fileclass	The protection class of a file. A protection class must be one of the following: A-Z, a-z, #, \$, @, or blank. This can be specified by a constant, a variable, a partial backward reference, or any other string expression.
filename	A one to eight character string consisting of the letters A through Z, @, #, \$, or 0 through 9 that identify a VS file. This can be specified by a constant, a variable, a partial backward reference, or any other string expression.
GETPARM	A request for parameter information by a program.
GOTO	Statement that specifies the next executable procedure language statement (performs either a forward or backward branch).
IF	Statement that controls conditional execution of procedure steps.
inner procedure	A procedure run from another procedure.
keyword	A word which identifies an operand within certain operand lists.
label	A string of one to eight alphanumeric characters that must begin with an alphabetic character which is used to identify a statement or certain clauses. A label may sometimes be the target of a GOTO statement, or used in a backward reference.
label-definition	A label-definition is a string of one to eight alphanumeric character that must begin with an alphabetic character and be immediately followed, with no intervening spaces, by a colon.
label-reference	A reference to a label-definition in a GOTO statement or a backward reference. The reference is written like the label-definition without the colon which follows.
libname	A one to eight character string consisting of the letters A through Z, @, #, \$, or 0 through 9 that identify a VS library. This can be specified by a constant, a variable, a partial backward reference, or any other string expression.

LOGOFF	Statement that terminates procedure execution and logs the user off the system.
MESSAGE	Statement that displays text on the workstation and continues execution of the procedure.
MOUNT	Statement that logically mounts a disk or tape volume.
nesting procedures	The technique of executing one or more procedures from another procedure.
operand	Data that is to be operated upon by the current procedure statement.
outer procedure	A procedure that executes another procedure.
owner	A string of one to three alphanumeric characters that must begin with an alphabetic character contain no embedded spaces. The string identifies the owner of record for a specified file or library.
partial backward reference	A backward reference to a specific parameter.
period	The retention period in days (0-999) for a file.
PRINT	Statement that places a print file in the PRINT Queue.
procedure	A routine which can run system or user functions and supply parameter information.
procedure user	Individual who runs a procedure.
procedure writer	Individual who writes a procedure.
PROMPT	Statement that displays variables and constants on the workstation and also allows the user to accept data for variables.
PROTECT	Statement that protects a disk file or library.
prname	A name that identifies a specific GETPARM request.
RENAME	Statement that renames a disk file or library.
RETURN	Statement that terminates procedure execution.
RUN	Statement that runs a program or another procedure.

SCRATCH	Statement that scratches a disk file or library.
SET	Statement that sets default values for file assignments, procedure submittal defaults, and print mode options.
statement	A syntactically valid combination of operands, constants, variables, expressions, substrings, and labels.substring which begins with a verb, which performs a specific function.
SUBMIT	Statement that submits a procedure file into the PROCEDURE Queue for noninteractive execution.
substring	A portion of a string variable defined by a "start" position for a specified "length."
TRACE	Statement that initiates or terminates tracing within a procedure.
USING	Statement that declares the formal parameters to a procedure.
variable	A storage area for information.
variable name	A string of 2 to 31 characters. The first character must be an ampersand (&). The other characters must be from the following: A-Z, a-z, 0-9, @, \$, #, and _.
verb	The portion of the procedure statement that describes the operation to be performed.
volname	A one to six character string consisting of the letters A through Z, @, #, \$, or 0 through 9 that identify a VS volume. This can be specified by a constant, a variable, a partial backward reference, or any other string expression.

INDEX

A

ALARM, 9-4
ASSIGN, 8-1, 12-3
Attributes, video, 9-2

B

Background processing, 5-7
Backward references, 2-6, 3-9
bell, 9-4
Boolean expressions, 2-7
Branching, 4-5
Built-in Functions, 2-5, 6-2

C

CANCEL EXIT (IS), 7-2
Conditional execution, 4-3
Constants, 2-3
Control
 flow of, 4-1
 of conditional execution, 4-3

D

DECLARE, 8-1, 12-6
Default, levels of, 3-2
DESTROY, 4-3, 12-9
DISMOUNT, 5-10, 12-10
DISPLAY, 3-4, 3-7

E

EDITOR, 1-4
 syntax checking from, 10-2
ENTER, 3-4, 3-5
ERASE, 9-3
ERROR EXIT (IS), 7-1
EXTRACT, 5-3, 12-12
Expressions, 2-3

F

Fields
 modifiable, 9-5
 multiple, 9-2
Flow of control, 4-1
Functions, 2-5, 6-2

G

GETPARM, 3-3
GOTO, 4-5, 12-15

I

IF, 4-3, 12-16
IF...Exists, 4-4
Interpreter, 1-1

K

Keywords, 3-4

L

Labels, 2-8
 multiple, 3-11
 referencing with, 3-8
 with GOTO, 4-6
LOGOFF, 5-10, 12-19

M

MESSAGE, 9-1, 12-20
Modifiable fields, 9-5
MOUNT, 5-10, 12-23
Multiple fields, 9-2

INDEX (continued)

N

Nested procedures, 3-11, 11-8

O

Operators, 2-6
 priority of, 2-7
 OPTIONS, 6-1, 12-25
 Options Screen, 1-3

P

Parameters, 7-3
 PRINT, 5-7, 12-26
 prname, 3-4
 PROC, 1-3
 syntax checking from, 10-4
 PROCEDURE, 12-28
 Procedures, 1-1
 entering and running, 1-3
 nesting level, 11-8
 PROCMSG, 6-1
 PROMPT, 9-1, 9-4, 12-29
 PROTECT, 5-6, 12-33

R

Referencing with labels, 3-8
 RENAME, 5-6, 12-36
 RETURN, 4-1, 12-38
 Return codes, A-1
 RUN, 12-39
 Running programs, 3-1

S

SCRATCH, 5-5, 12-43
 SET, 5-1, 12-45
 SUBMIT, 5-7, 12-47
 Substrings, 2-5
 Syntax, 1-4
 Syntax Checker, 10-1
 System Calls, 5-1

T

Tabs, 9-6
 TRACE, 11-2, 11-8, 12-51
 Tracing, 11-1
 from the EDITOR, 11-4
 from PROC, 11-6

U

Usage constants, 5-1
 USING, 7-4, 12-54

V

Variables, 2-4
 Verbs, 2-1
 Video attributes, 9-2
 VTOC, 5-5

&

&length, 6-2
 &time, 6-2
 &date, 6-2

WANG**Customer Comment Form**Publication Number **800-1205-**Title **VS PROCEDURE LANGUAGE REFEREN**

Help Us Help You . . .

We've worked hard to make this document useful, readable, and technically accurate. Did we succeed? Only you can tell us! Your comments and suggestions will help us improve our technical communications. Please take a few minutes to let us know how you feel.

How did you receive this publication?

- | | |
|--|--------------------------------------|
| ☐ Support or Sales Rep | ☐ Don't know |
| ☐ Wang Supplies Division | ☐ Other _____ |
| ☐ From another user | _____ |
| ☐ Enclosed with equipment | _____ |

How did you use this Publication?

- | | |
|--|--|
| ☐ Introduction to the subject | ☐ Aid to advanced knowledge |
| ☐ Classroom text (student) | ☐ Guide to operating instructions |
| ☐ Classroom text (teacher) | ☐ As a reference manual |
| ☐ Self-study text | ☐ Other _____ |

Please rate the quality of this publication in each of the following areas.

	EXCELLENT	GOOD	FAIR	POOR	VE PO
Technical Accuracy — Does the system work the way the manual says it does?	☐	☐	☐	☐	☐
Readability — Is the manual easy to read and understand?	☐	☐	☐	☐	☐
Clarity — Are the instructions easy to follow?	☐	☐	☐	☐	☐
Examples — Were they helpful, realistic? Were there enough of them?	☐	☐	☐	☐	☐
Organization — Was it logical? Was it easy to find what you needed to know?	☐	☐	☐	☐	☐
Illustrations — Were they clear and useful?	☐	☐	☐	☐	☐
Physical Attractiveness — What did you think of the printing, binding, etc?	☐	☐	☐	☐	☐

Were there any terms or concepts that were not defined properly? ☐ Y ☐ N If so, what were they? _____After reading this document do you feel that you will be able to operate the equipment/software? ☐ Yes ☐ No
☐ Yes, with practiceWhat errors or faults did you find in the manual? (Please include page numbers) _____

_____Do you have any other comments or suggestions? _____

Name _____ Street _____

Title _____ City _____

Dept/Mail Stop _____ State/Country _____

Company _____ Zip Code _____ Telephone _____

Thank you for your help.

WANG

0051510
OF 12151

Fold



**NO POSTAGE
NECESSARY
IF MAILED
IN THE
UNITED STATES**

BUSINESS REPLY CARD

FIRST CLASS

PERMIT NO. 16

LOWELL, MA

POSTAGE WILL BE PAID BY ADDRESSEE

**WANG LABORATORIES, INC.
TECHNICAL PUBLICATIONS
ONE INDUSTRIAL AVENUE
LOWELL, MASSACHUSETTS 01851**



Fold

WANG

The completed order form should be mailed to:

WANG LABORATORIES, INC.
 Supplies Division
 51 Middlesex St.
 No. Chelmsford MA 01863

To Order by Phone, Call:

(800) 225-0234

Hawaii, and A

(617) 256-1400**TELEX 951-743**

Order Form for Wang Manuals and Documentation

① Customer Number (If Known)				
② Bill To:			Ship To:	
③ Customer Contact:			④ Date	Purchase Order Number
() () Phone Name				
⑤ Taxable ⑥ Tax Exempt Number ⑦ Credit This Order to Yes ☐ A Wang Salesperson No ☐ Please Complete Salesperson's Name Employee No. RDB I				
⑧ Document Number	Description	Quantity	⑨ Unit Price	Total Price
Authorized Signature Date			Sub Total	
☐ Check this box if you would like a free copy of the Corporate Publications Literature Catalog (700-5294)			Less Any Applicable Discount	
			Sub Total	
			Local State Tax	
			Total Amount	

Ordering Instructions

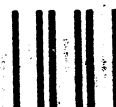
- | | |
|--|---|
| <ol style="list-style-type: none"> 1. If you have purchased supplies from Wang before, and know your Customer Number, please write it here. 2. Provide appropriate Billing Address and Shipping Address. 3. Please provide a phone number and name, should it be necessary for WANG to contact you about your order. 4. Your purchase order number and date. 5. Show whether order is taxable or not. 6. If tax exempt, please provide your exemption number | <ol style="list-style-type: none"> 7. If you wish credit for this order to be given to a WANG salesperson, please complete. 8. Show part numbers, description and quantity for each product ordered. 9. Pricing extensions and totaling can be completed at your option. Wang will refigure these prices and add freight on your invoice. 10. Signature of authorized buyer and date. |
|--|---|

Wang Supplies Division Terms and Conditions

- | | |
|--|--|
| <ol style="list-style-type: none"> 1. TAXES — Prices are exclusive of all sales, use, and like taxes. 2. DELIVERY — Delivery will be F O B. Wang's plant. Customer will be billed for freight charges; and unless customer specifies otherwise, all shipments will go best way surface as determined by Wang. Wang shall not assume any liability in connection with the shipment nor shall the carrier be construed to be an agent of Wang. If the customer requests that Wang arrange for insurance the customer will be billed for the insurance charges. | <ol style="list-style-type: none"> 3. PAYMENT — Terms are net 30 days from date of invoice. Unless otherwise stated by customer, partial shipments will generate partial invoices. 4. PRICES — The prices shown are subject to change without notice. Individual document prices may be found in the Corporate Publications Literature Catalog (700-5294) 5. LIMITATION OF LIABILITY — In no event shall Wang be liable for loss of data or for special, incidental or consequential damages in connection with or arising out of the use of or information contained in any manuals or documentation furnished hereunder. |
|--|--|

WANG

Fold



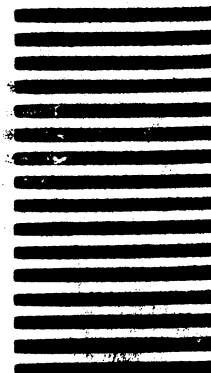
**NO POSTAGE
NECESSARY
IF MAILED
IN THE
UNITED STATES**

BUSINESS REPLY CARD

FIRST CLASS PERMIT NO. 16 NO. CHELMSFORD, MA.

POSTAGE WILL BE PAID BY ADDRESSEE

**WANG LABORATORIES, INC.
Supplies Division
c/o Order Entry Dept.
M/S 5511
51 Middlesex St.
No. Chelmsford, MA 01863**



Fold